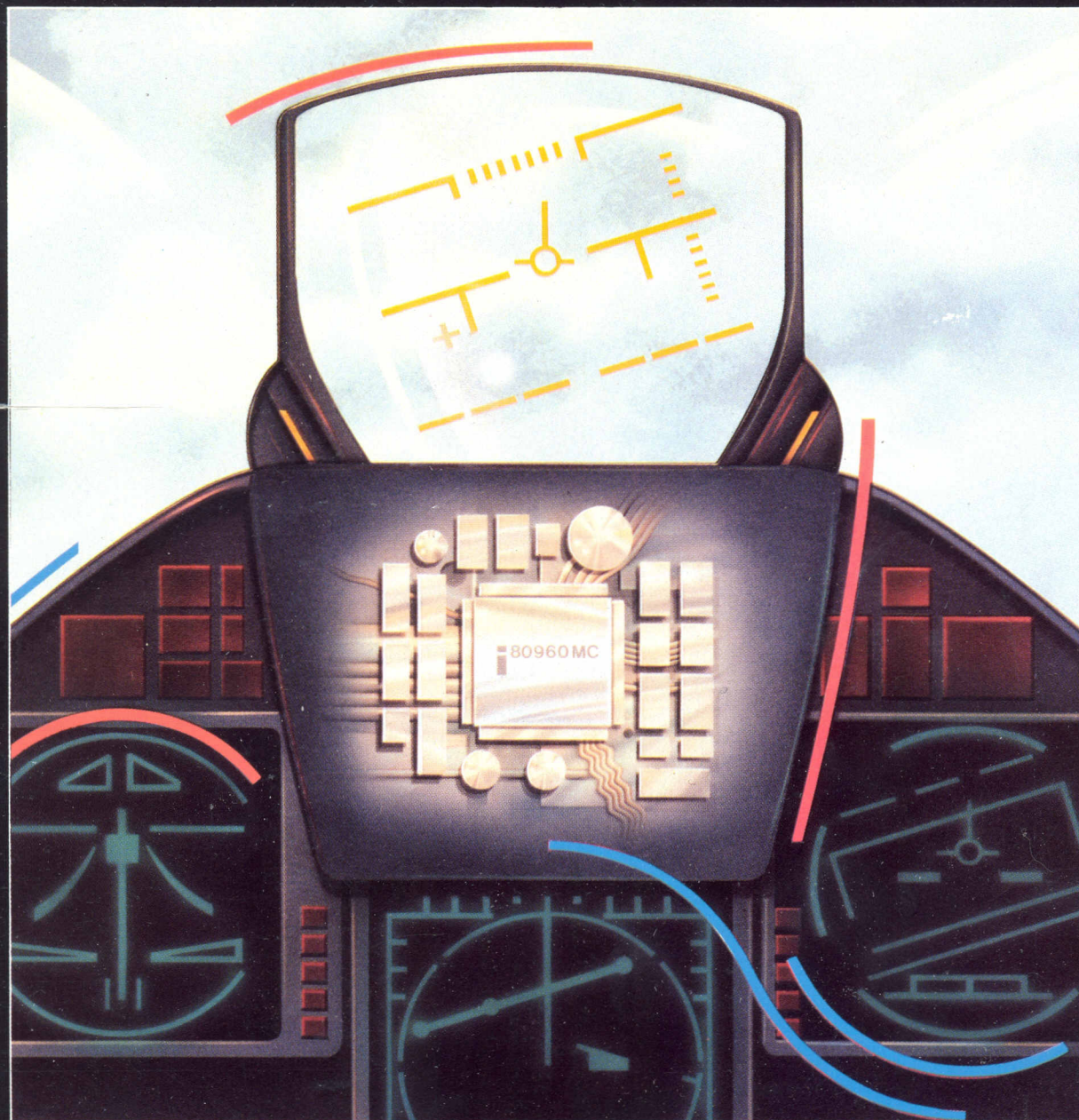


intel

# 80960MC Programmer's Reference Manual



Order Number: 271081-001



# 80960MC PROGRAMMER'S REFERENCE MANUAL

1988





Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel Products:

Above, BITBUS, COMMputer, CREDIT, Data Pipeline, FASTPATH, GENIUS, i, i<sup>2</sup>, ICE, ICEL, iCS, iDBP, iDIS, i<sup>2</sup>ICE, iLBX, i<sub>m</sub>, iMDDX, iMMX, Inboard, Insite, Intel, i<sub>tel</sub>, i<sub>tel</sub>BOS, Intel Certified, Inteleview, i<sub>tel</sub>ligent Identifier, i<sub>tel</sub>ligent Programming, Inteltec, Intellink, iOSP, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, KEPROM, Library Manager, MAP-NET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, PC-BUBBLE, Plug-A-Bubble, PROMPT, Promware, QUEST, QueX, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, SupportNET, UPI, and VLSiCEL, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix, 4-SITE.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

\*MULTIBUS is a patented Intel bus.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation  
Literature Distribution  
Mail Stop SC6-59  
3065 Bowers Avenue  
Santa Clara, CA 95051



## TABLE OF CONTENTS

<b>CHAPTER 1</b>	
<b>GUIDE TO THIS MANUAL</b>	
Manual Structure	1-1
Chapter Overview	1-1
Notation and Terminology	1-3
Reserved and Preserved	1-3
Set and Clear	1-4
<b>CHAPTER 2</b>	
<b>INTRODUCTION TO THE 80960 ARCHITECTURE</b>	
A New 32-Bit Architecture from Intel	2-1
High Performance Program Execution	2-2
Load and Store Model	2-2
On-Chip Caching of Code and Data	2-2
Overlapped Instruction Execution	2-3
Single-Clock Instructions	2-3
Efficient Interrupt Model	2-3
Simplified Programming Environment	2-4
Highly Efficient Procedure Call Mechanism	2-4
Versatile Instruction Set and Addressing	2-4
Extensive Fault Handling Capability	2-4
Debugging and Monitoring	2-5
Support for Architectural Extensions	2-5
Extensions Included in the 80960MC Processor	2-5
On-Chip Floating Point	2-5
String and Decimal Operations	2-6
Virtual-Memory Support	2-6
Protection	2-6
Multitasking	2-6
Multiprocessing	2-7
Fault Tolerance	2-7
Look for More in the Future	2-7
<b>CHAPTER 3</b>	
<b>EXECUTION ENVIRONMENT</b>	
Overview of the Execution Environment	3-1
Address Space	3-1
Register Model	3-2
Global Registers	3-4
Floating-Point Registers	3-4
Storage of Global and Floating-Point Registers	3-4



Local Registers	3-4
Register Alignment	3-4
Register Scoreboarding	3-5
Instruction Pointer	3-5
Arithmetic Controls	3-6
Initializing and Modifying the Arithmetic Controls	3-6
Functions of the Arithmetic-Controls Bits	3-7
Condition-Code Flags	3-7
Arithmetic-Status Flags	3-8
Integer-Overflow Flag and Mask	3-8
No-Imprecise-Faults Flag	3-9
Floating-Point Flags and Masks	3-9
Floating-Point-Normalizing-Mode Flag	3-9
Floating-Point-Rounding Control	3-9
Process and Trace Controls	3-10
Partitioning the Address Space	3-10
Instruction Caching	3-12
<b>CHAPTER 4</b>	
<b>PROCEDURE CALLS</b>	
Types of Procedure Calls	4-1
Call/Return Mechanism	4-1
Local Registers and the Procedure Stack	4-3
Procedure-Linking Information	4-3
Frame Pointer	4-3
Stack Pointer	4-3
Padding Area	4-5
Previous-Frame Pointer	4-5
Return Status and Prereturn-Trace Information	4-5
Return-Instruction Pointer	4-6
Mapping the Local Registers to the Procedure Stack	4-7
Local Call	4-8
Local-Call Operation	4-8
Local-Return Operation	4-8
Parameter Passing	4-9
Passing Parameters in Global Registers	4-9
Passing Parameters in an Argument List	4-9
Passing Parameters Through the Stack	4-9
System Call	4-10
Procedure Table	4-11
Procedure Entries	4-11
Supervisor-Stack Pointer	4-12
Trace-Control Flag	4-13
System Call to a Local Procedure	4-13

User-Supervisor Protection Model	4-13
User and Supervisor Modes	4-13
Supervisor Calls	4-14
Supervisor Stack	4-14
Branch and Link	4-15
<b>CHAPTER 5</b>	
<b>DATA TYPES AND ADDRESSING MODES</b>	
Data Types	5-1
Integers	5-1
Ordinals	5-1
Reals	5-2
Decimals	5-3
Bits and Bit Fields	5-4
Byte String	5-4
Triple and Quad Words	5-5
Byte, Word, and Bit Addressing	5-5
Literals	5-5
Register Addressing	5-6
Memory-Addressing Modes	5-6
Absolute	5-7
Register Indirect	5-7
Register Indirect with Index	5-7
Index with Displacement	5-7
IP with Displacement	5-7
<b>CHAPTER 6</b>	
<b>INSTRUCTION-SET SUMMARY</b>	
Instruction Formats	6-1
Assembly-Language Format	6-1
Machine Formats	6-1
Instruction Groups	6-2
Data Movement	6-4
Load	6-5
Store	6-5
Move	6-6
Load Address	6-6
Arithmetic	6-6
Add, Subtract, Multiply, and Divide	6-8
Extended Arithmetic	6-8
Remainder and Modulo	6-8
Shift and Rotate	6-9
Logical	6-10
Bit and Bit Field	6-10
Bit Operations	6-10



4-13	Bit-Field Operations	6-11
4-13	Byte Operations	6-11
4-14	Conversion	6-11
4-14	Comparison	6-11
4-15	Compare and Conditional Compare	6-11
	Compare and Increment or Decrement	6-12
	Branch	6-12
	Unconditional Branch	6-13
5-1	Conditional Branch	6-13
5-1	Compare and Branch	6-14
5-1	Test Condition Codes	6-14
5-2	Call and Return	6-15
5-3	Conditional Faults	6-15
5-4	Debug	6-16
5-4	Atomic Instructions	6-16
5-5	Processor Management	6-16
5-5	80960MC Non-Floating-Point Instruction-Set Extensions	6-17
5-5	Process Management	6-17
5-5	Process Control	6-17
5-5	Interprocess Communication	6-18
5-7	String	6-19
5-7	Decimal	6-19
5-7	Miscellaneous Instructions	6-20
5-7	Synchronous Load and Move	6-20
5-7	Memory-Management Functions	6-20
<b>CHAPTER 7</b>		
<b>FLOATING-POINT OPERATION</b>		
6-1	Introducing the 80960MC Floating-Point Architecture	7-1
6-1	Real Numbers and Floating-Point Format	7-1
6-1	Real Number System	7-1
6-2	Floating-Point Format	7-2
6-4	Normalized Numbers	7-3
6-5	Biased Exponent	7-4
6-5	Real Number and Non-Number Encodings	7-4
6-6	Signed Zeros	7-4
6-6	Signed, Nonzero, Finite Values	7-4
6-6	Denormalized Numbers	7-5
6-8	Signed Infinities	7-6
6-8	NaNs	7-6
6-8	Real Data Types	7-7
6-9	Execution Environment for Floating-Point Operations	7-7
6-10	Registers	7-8
6-10	Loading and Storing Floating-Point Values	7-9

8-11	Moving Floating-Point Values	7-10
8-11	Arithmetic Controls	7-11
8-11	Normalizing Mode	7-12
8-13	Rounding Control	7-12
8-14	Instruction Format	7-14
8-15	Instruction Operands	7-14
8-16	Summary of Floating-Point Instructions	7-15
8-16	Data Movement	7-15
8-18	Data-Type Conversion	7-15
8-18	Basic Arithmetic	7-17
8-19	Comparison, Branching, and Classification	7-17
8-20	Trigonometric	7-18
8-20	Pi	7-18
8-20	Logarithmic, Exponential, and Scale	7-19
8-21	Arithmetic Versus Nonarithmetic Instructions	7-20
8-21	Operations on NaNs	7-20
8-21	Exceptions and Fault Handling	7-21
8-22	Fault Handler	7-22
8-22	Floating-Reserved-Encoding Exception	7-22
8-24	Floating-Invalid-Operation Exception	7-23
8-24	Floating-Zero-Divide Exception	7-23
8-24	Floating-Overflow Exception	7-24
8-25	Floating-Underflow Exception	7-24
8-25	Floating-Inexact Exception	7-25
8-25	Floating-Point-Underflow Condition	7-26
8-26	<b>CHAPTER 8</b>	
8-26	<b>MEMORY MANAGEMENT</b>	
8-28	Introduction	8-1
8-28	Physical-Addressing Mode Versus Virtual-Addressing Mode	8-1
8-28	Physical Memory	8-2
	Physical-Memory Restrictions	8-2
	Caching of Memory Accesses	8-3
8-31	Virtual-Memory-Management System	8-3
8-31	Segment-Table Overview	8-4
8-32	Uses of Segments	8-5
8-32	Segment-Table Data Structures	8-6
8-32	Segment Selector	8-7
8-34	Segment Table	8-8
8-34	Segment Descriptors	8-9
8-35	Base Address	8-9
8-35	Size	8-10
8-35	Access Status	8-10
8-35	Valid Flag	8-10

7-10	Paging Method	8-11
7-11	Segment Types	8-11
7-12	Region Descriptors	8-11
7-13	Process, Port, and Procedure-Table Descriptors	8-13
7-14	Segment-Table Descriptors	8-14
7-14	Semaphore Descriptor	8-15
7-15	Invalid Descriptor	8-16
7-15	Page Tables and Page-Table Directories	8-16
7-15	Page Table and Page-Table-Directory Structure	8-18
7-17	Page Table and Page-Table-Directory Entries	8-18
7-17	Page-Table Entry	8-19
7-18	Page-Table-Directory Entry	8-20
7-18	Invalid Page Table or Page-Table-Directory Entry	8-20
7-19	Page Rights	8-20
7-20	Address Translation in Virtual Mode	8-21
7-20	SS Translation	8-21
7-21	Small Segment Table SS Translation	8-21
7-22	Large Segment Table SS Translation	8-22
7-22	Virtual-Address Translation	8-22
7-23	Simple-Region Address Translation	8-24
7-23	Paged-Region Address Translation	8-24
7-24	Bipaged Region-Address Translation	8-24
7-24	Load Physical Address Instruction	8-25
7-25	Spanning Page, Region, and Address-Space Boundaries	8-25
7-25	Translation Look-Aside Buffer	8-25
7-25	Operating-System Considerations	8-26
7-25	Address Space Structure	8-26
7-25	Region Gaps and Boundaries	8-28
7-25	Making Region Boundaries Transparent	8-28
7-25	Accessing System Data Structures	8-28
8-2	<b>CHAPTER 9</b>	
8-2	<b>PROCESSOR MANAGEMENT AND INITIALIZATION</b>	
8-3	Overview of Processor Configurations	9-1
8-4	Processes and Tasks	9-1
8-5	Processor-Management Facilities	9-2
8-6	Instruction List	9-2
8-7	System Data Structures	9-2
8-8	Interrupts	9-4
8-8	IACs	9-4
8-9	Faults	9-5
8-10	Process Scheduling and Dispatching	9-5
8-10	Processor-Control Block	9-5
8-10	Processor-Controls Word	9-5



8-01	System-Data-Structure Pointers	9-8
10-09	Miscellaneous PRCB Fields	9-9
10-10	Changing the PRCB	9-9
10-10	Priorities	9-10
10-10	Processor and Process States	9-10
10-12	Process-Executing and Process-Interrupted State	9-11
10-12	Stopped State	9-11
10-12	Idle and Idle-Interrupted States	9-11
10-13	Address-Translation Modes	9-12
10-13	Changing the Address-Translation Mode	9-12
	Processor Timing	9-13
	Duration of a Tick	9-13
	Idle Timing	9-13
11-1	Instruction Suspension	9-13
11-2	Software Requirements for Processor Management	9-14
11-2	Processor Initialization	9-15
11-3	Initial Memory Image	9-17
11-3	Check-Sum Words	9-17
11-3	Initialization Segment Table	9-17
11-3	Initialization PRCB	9-19
11-4	Initialization Code	9-19
11-5	Building a Memory Image	9-19
11-6	Typical Initialization Scenario	9-21
11-7	First Stage of Initialization	9-21
11-8	Second Stage of Initialization	9-23
11-9		
11-10	<b>CHAPTER 10</b>	
11-11	<b>INTERRUPTS</b>	
11-12	Overview of the Interrupt Facilities	10-1
11-13	Software Requirements for Interrupt Handling	10-1
11-14	Vectors and Priority	10-2
11-15	Interrupt Table	10-2
11-16	Interrupt-Table Sharing	10-4
11-17	Interrupt-Handler Procedures	10-4
11-18	Location of Interrupt Handler	10-4
11-19	Interrupt-Handler Restrictions	10-5
11-20	Interrupt Stack	10-5
11-21	Process Timing While Handling an Interrupt	10-6
11-22	Signaling Interrupts	10-6
11-23	Interrupts From Interrupt Pins	10-6
11-24	IAC Interrupts	10-7
11-25	System-Error Interrupt	10-8
	Interrupt-Handling Actions	10-8
	Receiving an Interrupt	10-8

9-8	Servicing an Interrupt	10-8
9-9	Process-Executing-State Interrupt	10-9
9-9	Process-Interrupted-State Interrupt	10-10
9-10	Interrupt Record	10-10
9-10	Idle-State Interrupt	10-10
9-11	Idle-Interrupted State Interrupt	10-12
9-11	Pending Interrupts	10-12
9-11	Posting Pending Interrupts	10-12
9-12	Checking for Pending Interrupts	10-13
9-12	Handling Pending Interrupts	10-13
9-13	<b>CHAPTER 11</b>	
9-13	<b>INTERAGENT COMMUNICATION</b>	
9-13	Introduction to IAC Messages	11-1
9-13	Software Requirement for Handling Internal IACs	11-1
9-14	Summary of IAC Messages	11-2
9-15	IAC-Message Format	11-2
9-17	Sending and Receiving an Internal IAC	11-3
9-17	Internal-IAC-Handling Action	11-3
9-19	IAC Faults	11-3
9-19	IAC-Message Reference	11-4
9-19	Check Process Notice	11-5
9-21	Continue Initialization	11-6
9-21	Flush Local Registers	11-7
9-23	Flush Process	11-8
9-23	Flush TLB	11-9
9-23	Flush TLB Page Table Entry	11-10
9-23	Flush TLB Physical Page	11-11
9-23	Flush TLB Segment Entry	11-12
9-23	Freeze	11-13
9-23	Interrupt	11-14
9-23	Modify Processor Controls	11-15
9-23	Preempt Process	11-16
9-23	Purge Instruction Cache	11-17
9-23	Reinitialize Processor	11-18
9-23	Restart Processor	11-19
9-23	Set Breakpoint Register	11-20
9-23	Stop Processor	11-21
9-23	Store Processor	11-22
9-23	Store System Base	11-23
9-23	Test Pending Interrupts	11-24
9-23	Warmstart Processor	11-25

<b>CHAPTER 12</b>	<b>FAULT HANDLING</b>	
Overview of the Fault-Handling Facilities		12-1
Fault Types		12-1
Fault-Handling Methods		12-3
Normal Fault-Handling Method		12-3
Overrides		12-4
System-Error Interrupt		12-4
Halt		12-5
Multiple Fault Conditions		12-5
Software Requirements for Handling Faults		12-5
Fault Table		12-5
Location of the Fault Table in Memory		12-7
Fault-Table Entries		12-7
Trace-Fault Handling		12-8
Fault-Handler Procedures		12-8
Possible Fault-Handler Actions		12-9
Process and Instruction Resumption Following a Fault		12-9
Returning With Resumption		12-10
Return Without Resumption		12-11
Aborting a Process		12-11
Fault Controls		12-12
Faults and Interrupts		12-13
Processing Timing While Handling a Fault		12-13
Generating a Fault		12-13
Fault-If and Mark Instructions		12-13
Event-Notice Fault		12-13
Fault Record		12-14
Saved Instruction Pointer		12-15
Resumption Record		12-15
Location of the Fault and Resumption Records		12-15
Fault-Handling Action		12-16
Selecting the Fault-Handling-Action Method		12-18
Normal Fault-Handling Action		12-18
Local Call/Return		12-18
Local Procedure-Table Call/Return		12-19
Supervisor Call/Return		12-19
Trace-Fault-Handler Call/Return		12-20
Override Fault-Handling Action		12-20
System-Error-Interrupt Action		12-21
Halt Action		12-22
Precise and Imprecise Faults		12-22
Fault Reference		12-24
Fault-Reference Notation		12-24

	Fault Type and Subtype	12-24
	Function	12-24
12-1	Fault Record	12-24
12-1	Saved IP	12-24
12-3	Process State Changes	12-24
12-3	Arithmetic Faults	12-25
12-4	Constraint Faults	12-26
12-4	Descriptor Faults	12-27
12-5	Event Faults	12-28
12-5	Floating-Point Faults	12-29
12-5	Machine Faults	12-31
12-5	Operation Faults	12-32
12-7	Process Faults	12-33
12-7	Protection Faults	12-34
12-8	Structural Faults	12-36
12-8	Trace Faults	12-37
12-9	Type Faults	12-39
12-9	Virtual-Memory Faults	12-40
12-10	<b>CHAPTER 13</b>	
12-11	<b>PROCESS MANAGEMENT</b>	
12-11	Process-Management Overview	13-1
12-12	Process Structure	13-1
12-13	Process State	13-1
12-13	Using Processes	13-2
12-13	Process-Control Block	13-2
12-13	Process Controls	13-4
12-14	Process-State Fields	13-5
12-15	Process Scheduling and Communication Fields	13-6
12-15	Process-Timing Fields	13-7
12-15	Storing of PCB Fields in the Processor	13-7
12-16	Changing the Process Controls	13-8
12-18	Changing the Arithmetic Controls	13-9
12-18	Changing the Process-Notice Field	13-9
12-18	Required Software Support for a Single-Process System	13-9
12-19	Physical Addressing Verses Virtual Addressing	13-9
12-19	Process Handling in a Single-Process System	13-11
12-20	<b>CHAPTER 14</b>	
12-20	<b>MULTIPLE-PROCESS MANAGEMENT</b>	
12-21	Overview of Multiple-Process-Management Facilities	14-1
12-22	Process Management Concepts	14-2
12-23	Scheduling and Dispatching	14-2
12-24	Process States	14-2
12-24	State-Transition Actions	14-3



Explicit Process-Dispatching .....	14-4
Process Timing .....	14-5
Time-Slice Scheduling .....	14-5
Execution-Time Counting .....	14-6
Overview of High-Level Process Management Facilities .....	14-6
Ports .....	14-7
FIFO Port .....	14-7
Priority Port .....	14-8
Message .....	14-9
Port Uses .....	14-9
Automatic Process Dispatching .....	14-10
Process-Scheduling Instructions .....	14-10
Process-Dispatching Action .....	14-10
Process Suspension .....	14-11
Process Synchronization .....	14-12
Use of Semaphores .....	14-12
Semaphore Structure .....	14-12
Semaphore-Handling Instructions .....	14-13
Semaphore-Access Actions .....	14-13
Process Preemption .....	14-14
Process-Preemption Action .....	14-15
Interprocess Communication .....	14-15
Communication Ports .....	14-15
Interprocess-Communication Mechanism .....	14-16
Send Message .....	14-16
Receive a Message .....	14-17
Send Service .....	14-18
Kernel Support for Message Passing .....	14-18
Applications of Messages .....	14-19
<b>CHAPTER 15</b> .....	
<b>MULTIPLE-PROCESSOR OPERATION</b> .....	
Overview of Multiprocessor-Support Facilities .....	15-1
External IAC Messages .....	15-1
Sending External IACs .....	15-1
Receiving and Handling External IACs .....	15-2
High-Level Process Management Facilities .....	15-3
Process Scheduling and Dispatching .....	15-4
Multiprocessor Preemption .....	15-4
Preemption Control .....	15-4
Multiprocessor-Preemption Action .....	15-6
Atomic Instructions .....	15-6
Interrupt Handling in a Multiprocessor System .....	15-7

<b>CHAPTER 16</b>	
<b>DEBUGGING</b>	
Overview of the Trace-Control Facilities	16-1
Required Software Support for Tracing	16-1
Trace Controls	16-1
Trace-Controls Word	16-2
Trace-Enable and Trace-Fault-Pending Flags	16-3
Trace Control on Supervisor Calls	16-3
Trace Modes	16-4
Instruction Trace	16-4
Branch Trace	16-4
Call Trace	16-4
Return Trace	16-5
Prereturn Trace	16-5
Supervisor Trace	16-5
Breakpoint Trace	16-5
Trace-Fault Handler	16-6
Signaling a Trace Event	16-6
Handling Multiple Trace Events	16-6
Trace-Handling Action	16-7
Normal Handling of Trace Events	16-7
Prereturn-Trace Handling	16-7
Tracing and Interrupt Handlers	16-7
Tracing and Fault Handlers	16-8
<b>CHAPTER 17</b>	
<b>INSTRUCTION REFERENCE</b>	
Introduction	17-1
Notation	17-1
Alphabetic Reference	17-2
Mnemonic	17-2
Format	17-2
Description	17-3
Action	17-3
Faults	17-3
Example	17-5
Opcode and Instruction Format	17-5
See Also	17-5
Instructions	17-5
<b>APPENDIX A</b>	
<b>INSTRUCTION AND DATA STRUCTURE QUICK REFERENCE</b>	
Instruction Quick Reference	A-1
Instruction List by Assembler Mnemonic	A-2
Instruction List by Opcode	A-7

Summary of System Data Structures	A-12
Execution Environment	A-12
Memory Management	A-15
Processor Management	A-20
Interrupt Handling	A-23
IACs	A-25
Fault Handling	A-25
Process Management	A-27
Trace Control	A-30
<b>APPENDIX B</b>	
<b>MACHINE-LEVEL INSTRUCTION FORMATS</b>	
General Instruction Format	B-1
REG Format	B-2
COBR Format	B-3
CTRL Format	B-4
MEM Format	B-4
MEMA Format Addressing	B-5
MEMB Format Addressing	B-6
<b>APPENDIX C</b>	
<b>INSTRUCTION TIMING</b>	
Introduction	C-1
Internal Structure of the 80960MC Processor	C-1
Memory Management Unit	C-1
Bus Control Logic	C-2
Instruction Fetch Unit and Instruction Cache	C-3
Instruction Decoder	C-4
Simple Instructions	C-4
Floating Point and Branch Instructions	C-4
Complex Instructions	C-5
Load and Store Instructions	C-5
Micro-Instruction Sequencer and ROM	C-6
Instruction Execution Unit	C-6
Register Bypassing	C-7
Floating Point Unit	C-8
Execution times for the 80960 Architecture Instructions	C-8
Logical Instructions	C-9
Bit Instructions	C-10
Register Moves	C-10
Integer and Ordinal Arithmetic	C-11
Multiply and Divide Instructions	C-12
Branching	C-12
Call/Return Instructions	C-13
Miscellaneous Complex Instructions	C-14

15-A	Load Instructions	C-14
15-A	Store Operations	C-16
15-A	Execution times for the Extended Instructions	C-16
20-A	Decimal Instructions	C-17
23-A	Floating-Point Instructions	C-17
25-A	Process-Management Instructions	C-17
25-A	<b>APPENDIX D</b>	
27-A	<b>INITIALIZATION CODE</b>	
30-A	Overview	D-1
	Example Code	D-2
	startup.s	D-4
1-B	f_table.lst	D-9
2-B	i_table.lst	D-10
3-B	initial_frame.lst	D-14
4-B	macs.m4	D-15
4-B	f_handle.c	D-16
5-B	i_handle.c	D-16
6-B	fix_pte.c	D-16
	prog1.c	D-18
	prog2.c	D-19
	led.h	D-19
1-C	pass1.ld	D-20
1-C	pass1a.ld	D-20
1-C	pass2.ld	D-21
2-C	<b>APPENDIX E</b>	
3-C	<b>CONSIDERATIONS FOR WRITING PORTABLE SOFTWARE</b>	
4-C	Architecture Restrictions	E-1
4-C	SALIGN Parameter	E-1
5-C	Boundary Alignment	E-2
5-C	Faults	E-2
6-C	Physical Memory	E-2
6-C	IACs	E-2
7-C	Timing	E-2
8-C	Interrupts	E-3
8-C	Initialization	E-3
9-C	Multiprocessor Preemption	E-3
10-C	Breakpoints	E-3
10-C	Implementation Dependent Instructions	E-3
11-C	Lock Pin	E-3
12-C		
12-C		
13-C		
14-C		



9-8	Algorithm for First Stage of Initialization	9-18
10-1	Interrupt Table	10-1
3-1	Execution Environment	3-2
3-2	Registers Available to a Single Procedure	3-3
3-3	Arithmetic Controls	3-6
3-4	Address Space Regions	3-10
3-5	Typical Use of Address-Space Regions	3-11
4-1	Local Registers and Procedure Stack	4-2
4-2	Procedure Stack Structure	4-4
4-3	System-Call Mechanism	4-10
4-4	Procedure-Table Structure	4-12
5-1	Integer Format and Range	5-2
5-2	Ordinal Format and Range	5-3
5-3	Decimal Format	5-4
5-4	Bits and Bit Fields	5-4
7-1	Binary Number System	7-2
7-2	Binary Floating-Point Format	7-3
7-3	Real Numbers and NaNs	7-5
7-4	Real-Number Formats	7-7
7-5	Storage of Real Values in Global and Local Registers	7-9
7-6	Interaction of Floating Underflow and Inexact Exceptions	7-27
8-1	Conceptual View of the Segment Table	8-4
8-2	Segment Addressing	8-5
8-3	Uses of Segments	8-6
8-4	Segment Selector	8-7
8-5	Segment Table	8-8
8-6	Generic Segment Descriptor	8-9
8-7	Region Segment Descriptors	8-12
8-8	Process, Port, and Procedure-Table Segment Descriptors	8-14
8-9	Segment-Table Segment Descriptors	8-15
8-10	Semaphore Segment Descriptor	8-15
8-11	Invalid Segment Descriptor	8-16
8-12	Conceptual View of Segment Paging	8-17
8-13	Page Table or Page-Table-Directory Structure	8-18
8-14	Page Table or Page-Table-Directory Entries	8-19
8-15	Virtual-Address Translation	8-23
8-16	Address Space Structure	8-27
8-17	Making Region Boundaries Transparent	8-29
8-18	Mapping of Physical Memory to Region 3	8-31
9-1	System Data Structures	9-3
9-2	Processor-Control Block (PRCB)	9-6
9-3	Processor-Controls Word	9-7
9-4	Required Fields in PRCB for Single-Task Configuration	9-16
9-5	Initial Memory Image	9-18

9-6. Algorithm for First Stage of Initialization Procedure .....	9-22
10-1. Interrupt Table .....	10-3
10-2. Interrupt-Control Register .....	10-6
10-3. Storage of an Interrupt Record on the Stack .....	10-11
11-1. IAC-Message Format .....	11-2
12-1. Fault Table and Fault-Table Entries .....	12-6
12-2. Fault Record .....	12-14
12-3. Storage of the Fault and Resumption Records on the Stack .....	12-17
13-1. Process-Control Block (PCB) .....	13-3
13-2. Process-Controls Word .....	13-4
13-3. Process Notice Field and Event-Fault Flags .....	13-7
13-4. Process-Control Block for Single-Process System .....	13-10
14-1. Process States .....	14-3
14-2. Ports .....	14-8
14-3. Queue Record .....	14-9
14-4. Semaphore Structure .....	14-13
15-1. Encoding of Address for Processor Receiving an IAC .....	15-2
16-1. Trace-Controls Word .....	16-2
A-1. Arithmetic Controls (Chapter 3) .....	A-12
A-2. Registers Available to a Single Procedure (Chapter 3) .....	A-13
A-3. Procedure Stack Structure (Chapter 4) .....	A-14
A-4. SS's, Segment Table, and Segments (Chapter 8) .....	A-15
A-5. Generic Segment Descriptor (Chapter 8) .....	A-15
A-6. Region Segment Descriptors (Chapter 8) .....	A-16
A-7. Process, Port, and Procedure Table Segment Descriptors (Chapter 8) .....	A-17
A-8. Segment-Table Segment Descriptors (Chapter 8) .....	A-17
A-9. Semaphore Segment Descriptor (Chapter 8) .....	A-18
A-10. Invalid Segment Descriptor (Chapter 8) .....	A-18
A-11. Page Table or Page-Table Directory Entries (Chapter 8) .....	A-19
A-12. Processor Controls (Chapter 9) .....	A-20
A-13. PRCB (Chapter 9) .....	A-21
A-14. Initial Memory Image (Chapter 9) .....	A-22
A-15. Interrupt Table (Chapter 10) .....	A-23
A-16. Interrupt Record on Stack (Chapter 10) .....	A-24
A-17. IAC Message Format (Chapter 11) .....	A-25
A-18. Fault Record (Chapter 12) .....	A-25
A-19. Fault Table and Fault-Table Entries (Chapter 12) .....	A-26
A-20. PCB (Chapter 13) .....	A-27
A-21. Process Controls (Chapter 13) .....	A-28
A-22. Ports (Chapter 14) .....	A-29
A-23. Trace Controls (Chapter 16) .....	A-30
B-1. Instruction Formats .....	B-1
C-1. Block Diagram of the 80960MC Processor .....	C-2
C-2. Execution Time of an Instruction .....	C-8

C-3. Load Where the Next Instruction Requires the Fetched Data	C-15
C-4. Load Where the Next Instruction Does Not Require the Fetched Data	C-15
C-5. Back-to-Back Load Instructions	C-16

## Tables

1-1. Chapters of Interest to Specific Users	1-1
3-1. Condition Codes for True or False Conditions	3-7
3-2. Condition Codes for Inequality Conditions	3-8
3-3. Encoding of Arithmetic-Status Field	3-8
3-4. Encoding of Floating-Point-Rounding-Control Field	3-9
4-1. Encoding of Return-Status Field	4-6
4-2. Encodings of Entry Type Field in Procedure Table Entry	4-11
5-1. Addressing Modes	5-6
6-1. Summary of the 80960 Instruction Set	6-3
6-2. Summary of the 80960MC Instruction-Set Extensions	6-4
6-3. Arithmetic Operations	6-7
7-1. Real-Number Notation	7-3
7-2. Denormalization Process	7-6
7-3. Real Numbers and NaN Encodings	7-8
7-4. Arithmetic Controls Used in Floating-Point Operations	7-11
7-5. Rounding Methods	7-13
7-6. Rounding of Positive Numbers	7-13
7-7. Rounding of Negative Numbers	7-13
7-8. Format of QNaN Results	7-21
8-1. Page Access Rights Interpretation	8-20
9-1. Encoding of the State Field	9-7
9-2. ROM and RAM Resident Data Structures	9-20
11-1. IAC Messages	11-2
12-1. Fault Types and Subtypes	12-2
12-2. Fault Flags or Masks	12-12
13-1. Encoding of the Process-State Field	13-5
B-1. Encoding of Src1 and Src2 Fields in REG Format	B-2
B-2. Encoding of Src/Dst Field in REG Format	B-3
B-3. Addressing Modes for MEM Format Instructions	B-5
B-4. Encoding of Scale Field	B-6
C-1. Registers Scoreboarded According to Registers Referenced	C-7
C-2. Logical Instruction Timing	C-9
C-3. Bit Instruction Timing	C-10
C-4. Scan and Span Bit Instruction Timing	C-10
C-5. Move Instruction Timing	C-10

C-6.	Integer and Ordinal Arithmetic Instruction Timing	C-11
C-7.	Compare Instruction Timing	C-11
C-8.	Multiply and Divide Instruction Timing	C-12
C-9.	Multiply/Divide Execution Times Based on Significant Bits	C-12
C-10.	Branch Instruction Timing	C-13
C-11.	Miscellaneous Complex Instruction Timing	C-14
C-12.	Decimal Instruction Timing	C-17
C-13.	Simple Floating-Point Instruction Timing	C-18
C-14.	Complex Floating-Point Instruction Timing	C-19
C-15.	Process-Management Instruction Timing	C-19
1-1.	Chapters of Interest to Specific Users	
3-1.	Condition Codes for True or False Conditions	
3-2.	Condition Codes for Inequality Conditions	
3-3.	Encoding of Arithmetic-Status Field	
3-4.	Encoding of Floating-Point-Rounding-Control Field	
4-1.	Encoding of Return-Status Field	
4-2.	Encoding of Entry Type Field in Procedure Table Entry	
5-1.	Addressing Modes	
6-1.	Summary of the 80960 Instruction Set	
6-2.	Summary of the 80960MC Instruction-Set Extensions	
6-3.	Arithmetic Operations	
7-1.	Real-Number Notation	
7-2.	Denormalization Process	
7-3.	Real Numbers and NaN Encodings	
7-4.	Arithmetic Controls Used in Floating-Point Operations	
7-5.	Rounding Methods	
7-6.	Rounding of Positive Numbers	
7-7.	Rounding of Negative Numbers	
7-8.	Format of QNaN Results	
8-1.	Page Access Rights Interpretation	
9-1.	Encoding of the State Field	
9-2.	ROM and RAM Resident Data Structures	
11-1.	IAC Messages	
12-1.	Fault Types and Subtypes	
12-2.	Fault Flags or Masks	
13-1.	Encoding of the Process-State Field	
B-1.	Encoding of Src1 and Src2 Fields in REG Format	
B-2.	Encoding of SrcDat Field in REG Format	
B-3.	Addressing Modes for MEM Format Instructions	
B-4.	Encoding of Scale Field	
C-1.	Registers Scoreboarded According to Registers Referenced	
C-2.	Logical Instruction Timing	
C-3.	Bit Instruction Timing	
C-4.	Scan and Span Bit Instruction Timing	
C-5.	Move Instruction Timing	



---

# *Guide to this Manual*

**1**

---

---

# Guide to this Manual

1

---

# CHAPTER 1

## GUIDE TO THIS MANUAL

This chapter describes the organization of this manual, the contents of each chapter, and terminology used in the manual. It also outlines the chapters of the manual that are of most interest to applications programmers, compiler designers, and designers of operating-system kernels (or system executives).

### MANUAL STRUCTURE

This manual is a reference manual for the Intel 80960MC processor. It gives programmers and system designers detailed information about the processor's programming environment and its operating-system-support facilities.

The book is divided into three parts. Chapters 2 through 7 describe the processor's programming environment, which includes the instruction-execution environment, data types, addressing modes, floating-point operations, and instruction set. Chapters 8 through 16 describe the facilities to support kernel functions, which include the memory management, processor management, interrupt handling, fault handling, process management, and debug facilities. Chapter 17 provides detailed descriptions of all the instructions in the instruction set, organized in alphabetical order.

Table 1-1 shows those chapters that will be of most interest to applications programmers, compiler designers, or kernel designers.

**Table 1-1: Chapters of Interest to Specific Users**

User	Chapters
Applications Programmer	Chapters 2 through 7; Chapter 17.
Compiler Designer	Chapters 2 through 7; Chapters 10, 12, and 17; and Appendices A, B, C, and E.
Kernel Designer	Chapters 2 through 17; and Appendices D and E.

### CHAPTER OVERVIEW

The following is a brief overview of the contents of each chapter:

**Chapter 1 — Guide to This Manual.** Overview of this manual.

**Chapter 2 — Introduction to the 80960 Architecture.** Overview of the Intel 80960 architecture, the architecture on which the 80960MC processor is based.

**Chapter 3 — Execution Environment.** Description of the environment in which instructions are executed. The topics discussed in this chapter include the address space, registers, instruction pointer, and arithmetic controls.

**Chapter 4 — Procedure Calls.** Description of the various mechanisms available for making procedure calls. The topics discussed here include the local call/return mechanism, procedure stack, branch-and-link procedure calls, procedure table calls, and supervisor/user protection model.

**Chapter 5 — Data Types and Addressing Modes.** Description of the non-floating-point data types and how bit and byte strings are addressed. The addressing modes provided for addressing data in memory are also described in this chapter.

**Chapter 6 — Instruction-Set Summary.** Overview of all the non-floating point instructions in the 80960MC instruction set, arranged by functional groups. Also included is a brief description of the assembly-language instruction format.

**Chapter 7 — Floating-Point Operation.** Description of the processor's floating-point processing facilities. This chapter includes an overview of floating-point numbers and a description of the 80960MC floating-point data types and their relationship to the IEEE floating-point standard. Descriptions of the floating-point instructions, exceptions, and faults are also included.

**Chapter 8 — Memory Management.** Description of the memory management facilities. The topics discussed here include the physical-memory requirements, physical addressing, and the virtual-memory management facilities.

**Chapter 9 — Processor Management and Initialization.** Description of the processor management facilities. Included is a discussion of the processor control block (PRCB), processor states, priorities, processor timing, and the software requirements for processor management. The requirements for processor initialization are described at the end of the chapter.

**Chapter 10 — Interrupts.** Description of the interrupt mechanism, interrupt priority, interrupt table, interrupt-handling procedures, and the software requirements for handling interrupts.

**Chapter 11 — Interagent Communication.** Description of the interprocessor communication (IAC) mechanism, which allows several processors to communicate with one another on the bus. The topics covered in this chapter include the IAC mechanism and software requirements for using internal IACs. A detailed description of each IAC is given in a reference section at the end of the chapter.

**Chapter 12 — Fault Handling.** Description of the processor's fault-handling mechanism. Included here is a discussion of the fault-table structure, fault-handling procedures, and the software requirements for handling faults. A detailed description of each fault is given in a reference section at the end of the chapter.

**Chapter 13 — Process Management.** Description of the process management facilities. The topics discussed here include the process control block (PCB) and the software requirements for running a single process.

**Chapter 14 — Multiple-Process Management.** Overview of the facilities provided to manage multiple processes. The topics discussed in this chapter include explicit process dispatching, process timing, automatic process dispatching, process synchronization, and interprocess communication.



**Chapter 15 — Multiple-Processor Operation.** Overview of the facilities to support multiple processor configurations. Included are descriptions of the external IAC handling mechanism, process preemption, and the atomic instructions.

**Chapter 16 — Debugging.** Description of the debugging and monitoring support facilities, including the trace control register.

**Chapter 17 — Instruction Reference.** Alphabetical listing of the complete 80960MC instruction set with detailed descriptions of each instruction, assembly-language syntax, examples, and algorithms.

**Appendix A — Instruction and Data Structure Quick Reference.** Two lists of the 80960MC instructions: one sorted alphabetically by assembly-language mnemonic and one sorted by machine language opcode. A collection of illustrations showing the system data structures is also provided here.

**Appendix B — Machine-Level Instruction Formats.** Description of the machine-level instruction formats.

**Appendix C — Instruction Timing.** Description of the 80960MC processor's instruction pipeline and how it affects instruction timing. The numbers of clock cycles required for each instruction are also given.

**Appendix D — Initialization Code.** Listing of sample code to initialize the 80960MC processor.

**Appendix E — Considerations for Writing Portable Software.** Discussion of various aspects of the 80960 architecture that should be considered if code written for the 80960MC processor is intended to be ported at a later date to other processors in the Intel 80960 family.

## NOTATION AND TERMINOLOGY

The following paragraphs describe the notation and terminology used in this manual that have special meaning.

### Reserved and Preserved

Certain fields in the processor's system data structures are described as being either *reserved* fields or *preserved* fields. A reserved field is one that is used by other implementations of the processor architecture. To help insure that a current software design is compatible with future processors based on the 80960 architecture, the bits in reserved fields should be set to 0 when the data structure is initially created. Thereafter, software should not access these fields.

Some fields in system data structures are shown as being required to be set to either 1 or 0. These fields should be treated as if they were reserved fields. They should be set to the specified value when the data structure is created and not accessed by software thereafter.

A preserved field is one that the processor does not use. Software may use preserved fields for any function.

## Set and Clear

The terms *set* and *clear* are used in this manual to refer to the value of a bit field in a system data structure. If a bit is set, its value is 1; if the bit is clear, its value is 0. Likewise, setting a bit means giving it a value of 1 and clearing a bit means giving it a value of 0.

## NOTATION AND TERMINOLOGY

The following paragraphs describe the notation and terminology used in this manual that have special meaning.

### Reserved and Preserved

Certain fields in the processor's system data structures are described as being either reserved fields or preserved fields. A reserved field is one that is used by other implementations of the processor architecture. To help insure that a current software design is compatible with future processors based on the 80960 architecture, the bits in reserved fields should be set to 0 when the data structure is initially created. Thereafter, software should not access these fields.

Some fields in system data structures are shown as being required to be set to either 1 or 0. These fields should be treated as if they were reserved fields. They should be set to the specified value when the data structure is created and not accessed by software thereafter.

---

*Introduction to the  
80960 Architecture*

---

**2**

---

# Introduction to the 80960 Architecture

---

2



## CHAPTER 2 INTRODUCTION TO THE 80960 ARCHITECTURE

This chapter provides an overview of the architecture on which the 80960MC processor is based.

### A NEW 32-BIT ARCHITECTURE FROM INTEL

The 80960MC processor is the military-grade member of a new family of processors from Intel. This processor family is based on a new 32-bit architecture called the 80960 architecture. The 80960 architecture has been designed specifically to meet the needs of embedded applications such as avionics, aerospace, weapons systems, robotics, and instrumentation, where high reliability is critical. It represents a renewed commitment from Intel to provide reliable, high-performance processors and controllers for the embedded processor marketplace.

The 80960 architecture can best be characterized as a high-performance computing engine. It features high-speed instruction execution and ease of programming. It is also easily extensible, allowing processors and controllers based on this architecture to be conveniently customized to meet the needs of specific processing and control applications.

Some of the important attributes of the 80960 architecture include:

- full 32-bit registers
- high-speed, pipelined instruction execution
- a convenient program execution environment with 32 general-purpose registers and a versatile set of special-function registers
- a highly optimized procedure call mechanism that features on-chip caching of local variables and parameters
- extensive facilities for handling interrupts and faults
- extensive tracing facilities to support efficient program debugging and monitoring
- register scoreboarding and write buffering to permit efficient operation with lower performance memory subsystems

The 80960MC processor implements the 80960 architecture, plus it offers several extensions to the architecture. Some of these extensions, such as on-chip support for floating-point arithmetic, virtual memory management, and multitasking, are designed to enhance overall system performance. Several other extensions are designed to enhance system reliability and robustness. These extensions include facilities for hardware enforced protection of software modules and for creating fault tolerant systems through the use of redundant processors.

The following sections describe those features of the 80960 architecture that are provided to streamline code execution and simplify programming. The extensions to this architecture provided in the 80960MC processor are described at the end of the chapter.

## HIGH PERFORMANCE PROGRAM EXECUTION

Much of the design of the 80960 architecture has been aimed at maximizing the processor's computational and data processing speed through increased parallelism. The following paragraphs describe several of the mechanisms and techniques used to accomplish this goal, including:

- an efficient load and store memory-access model
- caching of code and procedural data
- overlapped execution of instructions
- many one or two clock-cycle instructions

### Load and Store Model

One of the more important features of the 80960 architecture is that most of its operations are performed on operands in registers, rather than in memory. For example, all the arithmetic, logical, comparison, branching, and bit operations are performed with registers and literals.

This feature provides two benefits. First, it increases program execution speed by minimizing the number of memory accesses required to execute a program. Second, it reduces memory latency encountered when using slower, lower-cost memory parts.

To support this concept, the architecture provides a generous supply of general-purpose registers. For each procedure, 32 registers are available (28 of which are available for general use). These registers are divided into two types: global and local. Both these types of registers can be used for general storage of operands. The only difference is that global registers retain their contents across procedure boundaries, whereas the processor allocates a new set of local registers each time a new procedure is called.

The architecture also provides a set of fast, versatile load and store instructions. These instructions allow burst transfers of 1, 2, 4, 8, 12, or 16 bytes of information between memory and the registers.

### On-Chip Caching of Code and Data

To further reduce memory accesses, the architecture offers two mechanisms for caching code and data on chip: an instruction cache and multiple sets of local registers. The instruction cache allows prefetching of blocks of instruction from memory, which helps insure that the instruction execution pipeline is supplied with a steady stream of instructions. It also reduces the number of memory accesses required when performing iterative operations such as loops. (The size of the instruction cache can vary. With the 80960MC processor, it is 512 bytes.)

To optimize the architecture's procedure call mechanism, the processor provides multiple sets of local registers. This allows the processor to perform most procedure calls without having to write the local registers out to the stack in memory.

(The number of local-register sets provided depends on the processor implementation. The 80960MC processor provides four sets of local registers.)

### Overlapped Instruction Execution

Another technique that the 80960 architecture employs to enhance program execution speed is overlapping the execution of some instructions. This is accomplished through two mechanisms: register scoreboarding and branch prediction.

Register scoreboarding permits instruction execution to continue while data is being fetched from memory. When a load instruction is executed, the processor sets one or more scoreboard bits to indicate the target registers to be loaded. After the target registers are loaded, the scoreboard bits are cleared. While the target registers are being loaded, the processor is allowed to execute other instructions that do not use these registers. The processor uses the scoreboard bits to insure that target registers are not used until the loads are complete. (The checking of scoreboard bits is transparent to software.) The net result of using this technique is that code can often be optimized in such a way as to allow some instructions to be executed parallel.

### Single-Clock Instructions

It is the intent of the 80960 architecture that a processor be able to execute commonly used instructions such as move, add, subtract, logical operations, compare and branch in a minimum number of clock cycles (preferable one clock cycle). The architecture supports this concept in several ways. For example, the load and store model described earlier in this chapter (with its concentration on register-to-register operations) allows simple operations to be performed without the overhead of memory-to-memory operations.

Also, all the instructions in the 80960 architecture are 32 bits or 64 bits long and aligned on 32-bit boundaries. This feature allows instructions to be decoded in one clock cycle. It also eliminates the need for an instruction-alignment stage in the pipeline.

The design of the 80960MC processor takes full advantage of these features of the architecture, resulting in more than 50 instructions that can be executed in a single clock-cycle.

### Efficient Interrupt Model

The 80960 architecture provides an efficient mechanism for servicing interrupts from external sources. To handle interrupts, the processor maintains an interrupt table of 248 interrupt vectors (240 of which are available for general use). When an interrupt is signaled, the processor uses a pointer from the interrupt table to perform an implicit call to an interrupt handler procedure. In performing this call, the processor automatically saves the state of the processor prior to receiving the interrupt; performs the interrupt routine; and then restores the state of the processor. A separate interrupt stack is also provided to segregate interrupt handling from application programs.

The interrupt handling facilities also feature a method of prioritizing interrupts. Using this technique, the processor is able to store interrupts that are lower in priority than the task the processor is currently working on in a pending interrupt section of the interrupt table. At certain defined times, the processor checks the pending interrupts and services them.

### **SIMPLIFIED PROGRAMMING ENVIRONMENT**

Partly as a side benefit of its streamlined execution environment and partly by design, processors based on the 80960 architecture are particularly easy to program. For example, the large number of general-purpose registers allows relatively complex algorithms to be executed with a minimum number of memory accesses. The following paragraphs describe some of the other features that simplify programming.

#### **Highly Efficient Procedure Call Mechanism**

The procedure call mechanism makes procedure calls and parameter passing between procedures simple and compact. Each time a call instruction is issued, the processor automatically saves the current set of local registers and allocates a new set of local registers for the called procedure. Likewise, on a return from a procedure, the current set of local registers is deallocated and the local registers for the procedure being returned to are restored. On a procedure call, the program thus never has to explicitly save and restore those local variables and parameters that are stored in local registers.

#### **Versatile Instruction Set and Addressing**

The selection of instructions and addressing modes also simplifies programming. The architecture offers a full set of load, store, move, arithmetic, comparison, and branch instructions, with operations on both integer and ordinal data types. It also provides a complete set of Boolean and bit-field instructions, to simplify operations on bits and bit strings.

The addressing modes are efficient and straightforward, while at the same time providing the necessary indexing and scaling modes required to address complex arrays and record structures.

The large 4-gigabyte address space provides ample room to store programs and data. The availability of 32 addressing lines allows some address lines to be memory-mapped to control hardware functions.

#### **Extensive Fault Handling Capability**

To aid in program development, the 80960 architecture defines a wide selection of faults that the processor detects, including arithmetic faults, invalid operands, invalid operations, and machine faults. When a fault is detected, the processor makes an implicit call to a fault handler routine, using a mechanism similar to that described above for interrupts. The information collected for each fault allows program developers to quickly correct faulting code. It also allows automatic recovery from some faults.

## Debugging and Monitoring

To support debugging systems, the 80960 architecture provides a mechanism for monitoring processor activity by means of trace events. The processor can be configured to detect as many as seven different trace events, including branches, calls, supervisor calls, returns, prereturns, breakpoints, and the execution of any instruction. When the processor detects a trace event, it signals a trace fault and calls a fault handler. Intel provides several tools that use this feature, including an in-circuit emulator (ICE) device.

## SUPPORT FOR ARCHITECTURAL EXTENSIONS

The 80960 architecture described earlier in this chapter provides a high-performance computing engine for use as the computational and data-processing core of embedded processors or controllers. The architecture also provides several features that enable processors based on this architecture to be easily customized to meet the needs of specific embedded applications, such as signal processing, array processing, or graphics processing. The most important of these features is a set of 32 special-function registers. These registers provide a convenient interface to circuitry in the processor or to pins that can be connected to external hardware. They can be used to control timers, to perform operations on special data types, or to perform I/O functions.

The special-function registers are similar to the global registers. They can be addressed by all the register-access instructions.

## EXTENSIONS INCLUDED IN THE 80960MC PROCESSOR

The extensions to the 80960 architecture included in the 80960MC processor are built on top of the processor's core computing engine. These extensions are aimed at improving the efficiency and reliability of embedded systems.

## On-Chip Floating Point

The 80960MC processor provides a complete implementation of the IEEE standard for binary floating-point arithmetic (IEEE 754-185). This implementation includes a full set of floating-point operations, including add, subtract, multiply, divide, trigonometric functions, and logarithmic functions. These operations are performed on single precision (32-bit), double precision (64-bit), and extended precision (80-bit) real numbers.

One of the benefits of this implementation is that the floating-point handling facilities are completely integrated into the normal instruction execution environment. Single- and double-precision floating-point values are stored in the same registers as non-floating point values. Also, four 80-bit floating-point registers are provided to hold extended-precision values.



## String and Decimal Operations

The 80960MC processor provides several instructions for moving, filling, and comparing byte strings in memory. These instructions speed up string operations and reduce the amount of code required to handle strings.

The decimal instructions perform move, add with carry, and subtract with carry operations on binary-coded decimal (BCD) strings.

## Virtual-Memory Support

Another of the 80960MC processor's important features is support for virtual-memory management. When using the processor in virtual-memory mode, the processor provides each process (or task) with an address space of up to  $2^{32}$  bytes. This address space is paged into physical memory in 4K-byte pages. On-chip memory-management facilities handle virtual-to-physical address translation. A translation look-aside buffer (TLB) speeds address translation by storing virtual-to-physical address translations for frequently accessed parts of memory, such as the location of the page tables and the location of often used system data structures.

## Protection

The 80960MC processor offers two mechanisms for protecting critical data structures or software modules. The first is the ability to use page rights bits to restrict access to individual pages. Page rights allow various levels of access to be assigned to a page, ranging from no access to read only to read-write.

The second protection mechanism is a user/supervisor protection model. This two-level protection model provides hardware enforced protection of kernel procedures and data structures. When using this protection mechanism, privileged procedures and data are placed in protected pages of memory. These pages can then be accessed only through a procedure table, which provides a tightly controlled interface to kernel functions.

## Multitasking

The 80960MC processor offers a variety of process management facilities to support concurrent execution of multiple tasks. These facilities can be divided into two groups: process scheduling and interprocess communications.

The process scheduling facilities consist of a set of general-purpose data structures and instructions, which are designed to support several different multitasking schemes. For example, the processor provides a set of instructions that allow the kernel to explicitly dispatch a task (bind it to the processor) and to suspend a task (save the current state of a task so that another task can be bound to the processor). These instructions can be used within kernel procedures to schedule, dispatch, and preempt multiple tasks.

The processor also provides a unique feature called *self dispatching*. Here, the kernel schedules tasks by queuing them to a dispatch port. Thereafter, the processor handles the

dispatching, preempting, and rescheduling of the tasks automatically, independent of the kernel. When using this mechanism, tasks can be scheduled by priority, with up to 32 priority levels to choose from.

The processor's interprocess communication facilities include support for semaphores and communication ports. These facilities allow synchronization of interdependent tasks and asynchronous communication between tasks.

## **Multiprocessing**

The 80960MC processor provides several mechanisms designed to simplify the design of multiple-processor systems, allowing several processors to run in parallel, using shared memory resources. One of these mechanisms is the self-dispatching capability described above. Here, two or more processors can schedule and dispatch processes from a single dispatch port, with each processor equally sharing the processing load.

The processor also provides an interagent communication (IAC) mechanism that allows processors to exchange messages among themselves on the bus. This mechanism operates similarly to the interrupt mechanism, except that IAC messages are passed through dedicated sections of memory. The IAC mechanism can be used to preempt processes running on another processor, to manage interrupt handling, or to initialize and synchronize several processors.

A set of atomic instructions are also provided to synchronize memory accesses. Multiple processors can then access shared memory without inserting inaccuracies and ambiguities into shared data structures.

## **Fault Tolerance**

The 80960 family of components supports fault-tolerant system design through the use of the M82965 Bus Extension Unit component. The M82965 allows two processors to be operated in tandem to form a self-checking module. The two M82965s check the outputs of two processors (a master and a checker) cycle-by-cycle. If the checking M82965 detects a difference between outputs, it signals an error. A software recovery procedure can then be initiated.

This fault detection mechanism supports several fault detection and recovery techniques, including self healing, and continuous-operation (non-stop) systems.

## **LOOK FOR MORE IN THE FUTURE**

The 80960 architecture offers exceptional performance, plus a wealth of useful features to help in the design of efficient and reliable embedded systems. But equally important, it offers lots of room to grow. The 80960MC processor provides average instruction processing rates of 7.5 million instructions per second (7.5 MIPS) at 20 MHz clock rate and 10 MIPS at a 25 MHz clock rate<sup>1</sup>. This performance places the 80960MC at the top of the performance range for advanced, VLSI processor architectures.

---

<sup>1</sup>1 MIP is equivalent to the performance of a Digital Equipment Corp. VAX 11/780.

However, the 80960MC is only the beginning. With improvements in VLSI technology, future implementations of the 80960 architecture will offer even greater performance. They will also offer a variety of useful extensions to solve specific control and monitoring needs in the field of embedded applications.

The processor's interprocess communication facilities include support for semaphores and communication ports. These facilities allow synchronization of independent tasks and asynchronous communication between tasks.

## Multiprocessing

The 80960MC processor provides several mechanisms designed to simplify the design of multiple-processor systems, allowing several processors to run in parallel, using shared memory resources. One of these mechanisms is the self-dispatching capability described above. Here, two or more processors can schedule and dispatch processes from a single dispatch port, with each processor equally sharing the processing load.

The processor also provides an interagent communication (IAC) mechanism that allows processors to exchange messages among themselves on the bus. This mechanism operates similarly to the interrupt mechanism, except that IAC messages are passed through dedicated sections of memory. The IAC mechanism can be used to preempt processes running on another processor, to manage interrupt handling, or to initialize and synchronize several processors.

A set of atomic instructions are also provided to synchronize memory accesses. Multiple processors can then access shared memory without inserting inaccuracies and ambiguities into shared data structures.

## Fault Tolerance

The 80960 family of components supports fault-tolerant system design through the use of the M83965 Bus Extension Unit component. The M83965 allows two processors to be operated in tandem to form a self-checking module. The two M83965s check the outputs of two processors (a master and a checker) cycle-by-cycle. If the checking M83965 detects a difference between outputs, it signals an error. A software recovery procedure can then be initiated.

This fault detection mechanism supports several fault detection and recovery techniques, including self-healing, and continuous-operation (non-stop) systems.

## LOOK FOR MORE IN THE FUTURE

The 80960 architecture offers exceptional performance, plus a wealth of useful features to help in the design of efficient and reliable embedded systems. But equally important, it offers lots of room to grow. The 80960MC processor provides average instruction processing rates of 7.5 million instructions per second (7.5 MIPS) at 20 MHz clock rate and 10 MIPS at a 25 MHz clock rate. This performance places the 80960MC at the top of the performance range for advanced VLSI processor architectures.

1 MIPS is equivalent to the performance of a Digital Equipment Corp. VAX 11/780.







## CHAPTER 3

### EXECUTION ENVIRONMENT

This chapter describes how the 80960MC processor executes instructions and how it stores and manipulates data. The parts of the execution environment that are discussed include the address space, the register model, the instruction pointer, and the arithmetic controls.

The execution environment's procedure stack and procedure-call mechanism are described in Chapter 4.

#### OVERVIEW OF THE EXECUTION ENVIRONMENT

When a process (or a program running within the context of a process) is run on the 80960MC processor, the processor first sets up an execution environment for that process. It then begins executing instructions for that process, using this execution environment to store and manipulate data.

Figure 3-1 shows the part of the execution environment that the processor sets up to run a single procedure within a process. This environment consists of a  $2^{32}$ -byte address space, a set of global and floating-point registers, a set of local registers, a set of arithmetic-controls bits, the instruction pointer, a set of process-controls bits, and a set of trace-controls bits. All of these items reside on the 80960MC chip except the address space.

When the instruction stream for the process includes a procedure call, a procedure stack and some additional elements are added to this execution environment. These procedure-call related elements are shown and discussed in Chapter 4.

#### ADDRESS SPACE

Each process running on the processor is assigned a separate address space. From the point of view of the processor, this address space is flat (unsegmented) and byte addressable, with addresses running contiguously from 0 to  $2^{32} - 1$ . The process can allocate space for data, instructions, and the stack anywhere within this space.

The address space being described here is a logical address space that the operating system can map into physical memory either directly or indirectly (using the processor's virtual-addressing mechanism). The memory mapping method used is immaterial to this discussion. Once a process has been bound to the processor, the processor sees only the logical address space for that process.

#### NOTE

The memory-management method that the operating system uses can place some minor limitations on how the address space may be allocated. These limitations are described later in this chapter in the section titled "Partitioning the Address Space."

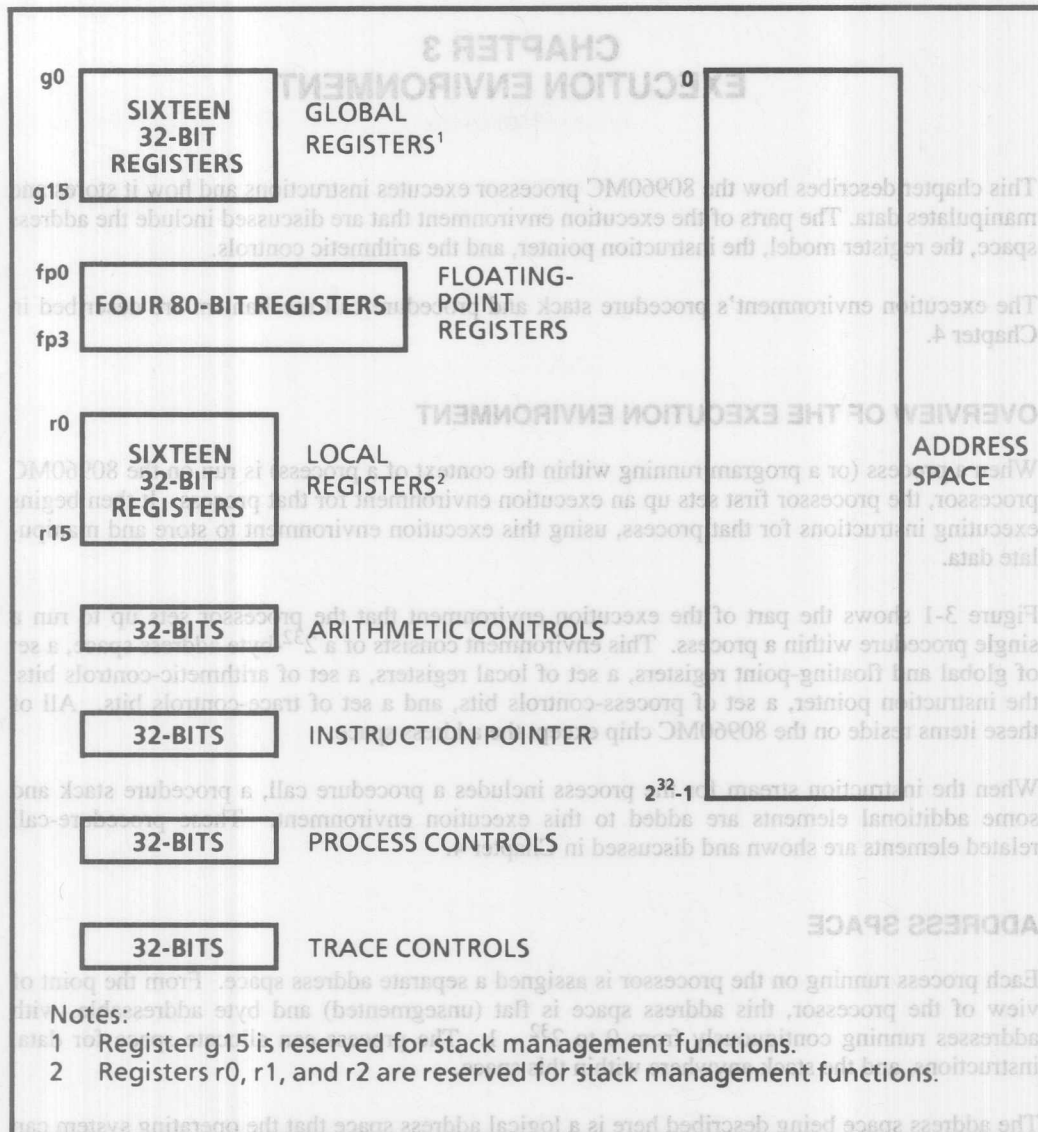


Figure 3-1: Execution Environment

## REGISTER MODEL

The processor provides three types of data registers: global, floating-point, and local. The 16 global registers constitute a set of general-purpose registers, the contents of which are preserved across procedure boundaries. The 4 floating-point registers are provided to support extended floating-point arithmetic. Their contents are also preserved across procedure boundaries. The 16 local registers are provided to hold parameters specific to a procedure (i.e.,

local variables). For each procedure that is called, the processor allocates a separate set of 16 local registers.

For any one procedure within a process, 36 registers are thus available (as shown in Figure 3-2): the 16 global registers, the 4 floating-point registers, and the 16 local registers. All of these registers are maintained on the processor chip.

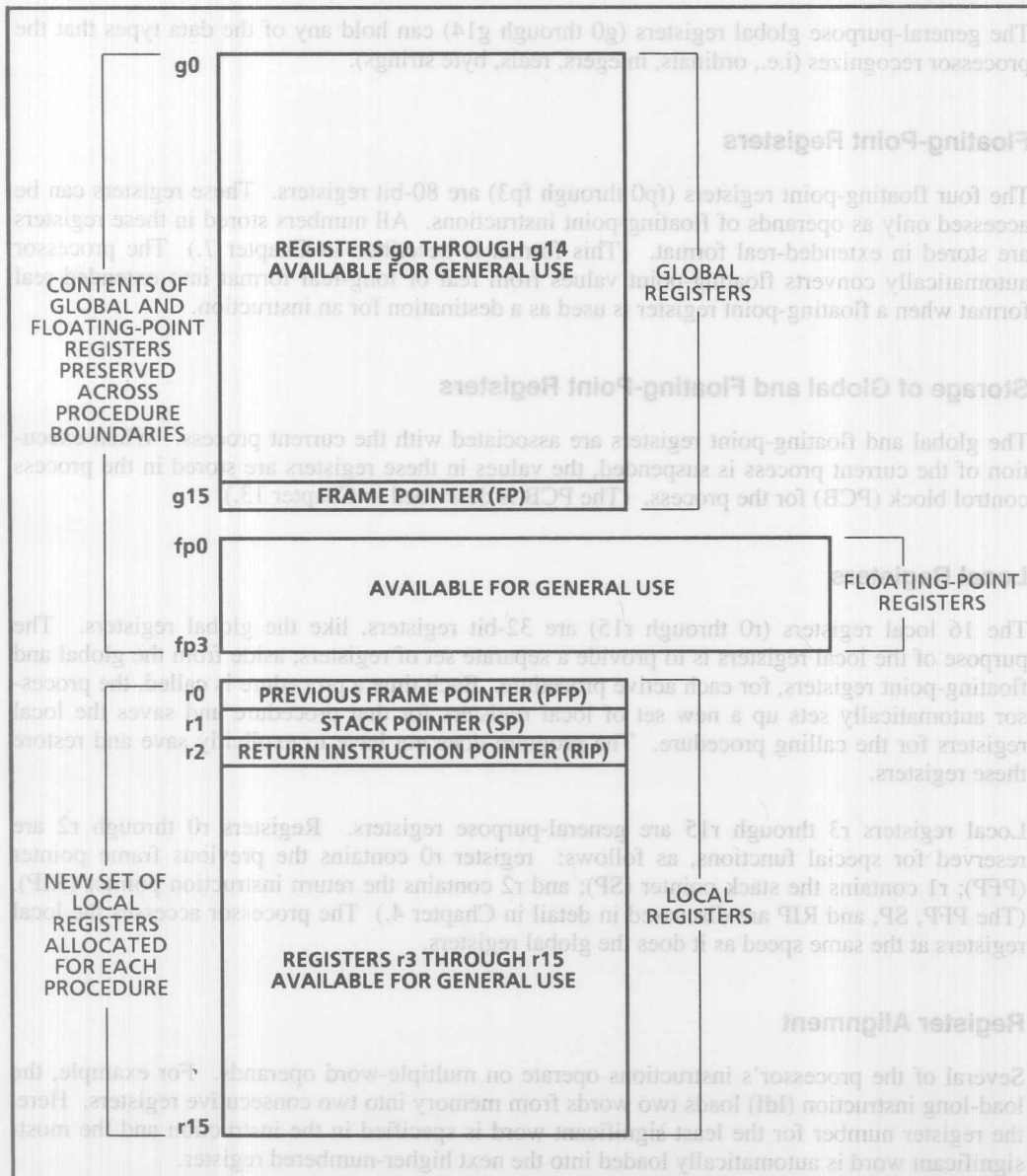


Figure 3-2: Registers Available to a Single Procedure

## Global Registers

The 16 global registers (g0 through g15) are 32-bit registers. Each register can thus hold a word (32 bits) of data. Registers g0 through g14 are general-purpose registers; g15 is reserved for the current frame pointer (FP). The FP contains the address of the first byte in the current (topmost) stack frame. (The FP and the procedure stack are discussed in detail in Chapter 4.)

The general-purpose global registers (g0 through g14) can hold any of the data types that the processor recognizes (i.e., ordinals, integers, reals, byte strings).

## Floating-Point Registers

The four floating-point registers (fp0 through fp3) are 80-bit registers. These registers can be accessed only as operands of floating-point instructions. All numbers stored in these registers are stored in extended-real format. (This format is described in Chapter 7.) The processor automatically converts floating-point values from real or long-real format into extended-real format when a floating-point register is used as a destination for an instruction.

## Storage of Global and Floating-Point Registers

The global and floating-point registers are associated with the current process. When execution of the current process is suspended, the values in these registers are stored in the process control block (PCB) for the process. (The PCB is described in Chapter 13.)

## Local Registers

The 16 local registers (r0 through r15) are 32-bit registers, like the global registers. The purpose of the local registers is to provide a separate set of registers, aside from the global and floating-point registers, for each active procedure. Each time a procedure is called, the processor automatically sets up a new set of local registers for that procedure and saves the local registers for the calling procedure. The program does not have to explicitly save and restore these registers.

Local registers r3 through r15 are general-purpose registers. Registers r0 through r2 are reserved for special functions, as follows: register r0 contains the previous frame pointer (PFP); r1 contains the stack pointer (SP); and r2 contains the return instruction pointer (RIP). (The PFP, SP, and RIP are discussed in detail in Chapter 4.) The processor accesses the local registers at the same speed as it does the global registers.

## Register Alignment

Several of the processor's instructions operate on multiple-word operands. For example, the load-long instruction (**ldl**) loads two words from memory into two consecutive registers. Here, the register number for the least significant word is specified in the instruction and the most-significant word is automatically loaded into the next higher-numbered register.

In cases where an instruction specifies a register number and multiple, consecutive registers are implied, the register number must be even if two registers are accessed (e.g., g0, g2) and an integral multiple of four if three or four registers are accessed (e.g., g0, g4). If a register reference for a source value is not properly aligned, the value is undefined. If a register reference for a destination value is not properly aligned, the registers that the processor writes to are undefined.

### Register Scoreboarding

The 80960MC provides a mechanism called *register scoreboarding* that in certain situations permits instructions to be executed concurrently. This mechanism works as follows. While an instruction is being executed, the processor sets a scoreboard bit to indicate that a particular register or group of registers is being used in an operation. If the instructions that follow do not use registers in that group, the processor in some instances is able to execute those instructions before execution of the prior instruction is complete. In effect, the register scoreboarding mechanism allows some instructions to be executed in parallel.

A common application of this feature is to execute one or more fast instructions (instructions that take one to three clock cycles) concurrently with load instructions. A load instruction typically takes 3 to 9 clock cycles (depending on the design of system memory and the addressing mode used). Register scoreboarding allows other instructions to be executed concurrently with the load instruction, providing that the other instructions do not affect the registers being loaded. For example, the following group of instructions loads a group of local registers while performing some other operations on data in global registers.

```
ld xyz, r6          # r6 ← data from address xyz
addi g4, g6, g7      # g7 ← g4 + g6
addi g9, g10, g11    # g11 ← g9 + g10
ld abc, r8           # r6 ← data from address abc
and g0, 0xffff, g1   # g1 ← g0 AND 0xffff
addi r6, r8, r7       # r7 ← r6 + r8
```

Here, the two **addi** instructions following the first load and the **and** instruction following the second load are performed concurrently with the bus accesses of the two load instructions. (Appendix C provides a detailed discussion of the processor's instruction-execution pipeline and register scoreboarding.)

### INSTRUCTION POINTER

The instruction pointer (IP) is the address (in the address space of the current process) of the instruction currently being executed. This address is 32 bits; however, since instructions are required to be aligned on word boundaries in memory, the 2 least-significant bits of the IP are always zero.

The IP is stored in the processor and cannot be read directly. However, the IP-with-displacement addressing mode allows the IP to be used as an offset into the address space. This addressing mode can also be used with the **lda** (load address) instruction to read the current value of the IP.



When a break occurs in the execution of a program or process (due to an interrupt, procedure call, or process suspension action), the IP of the next instruction to be executed (i.e., the RIP) is stored in local register r2, which is then stored on the stack. Refer to Chapter 4 for further discussion of this operation.

## ARITHMETIC CONTROLS

The processor's arithmetic controls are made up of a set of 32 bits, which are cached on the processor chip in the arithmetic-controls register. Figure 3-3 shows the arrangement of the arithmetic controls bits. The arithmetic controls bits include condition code flags; floating-point control and status flags and masks; integer control and status flags; and a flag that controls faulting on imprecise faults.

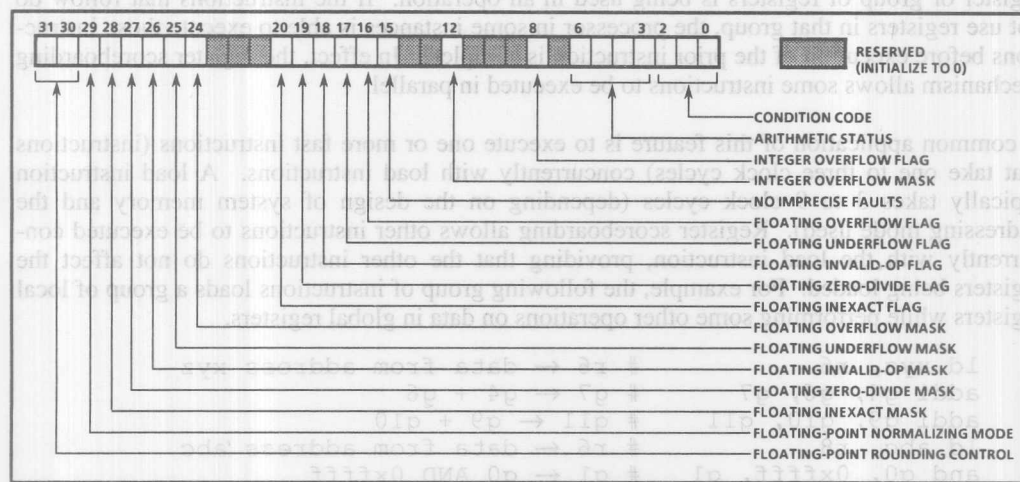


Figure 3-3: Arithmetic Controls

The processor sets or clears these bits to show the results of certain operations. For example, the processor modifies the condition code flags after each comparison operation to show the result of the comparison. Other arithmetic control bits, such as the floating-point fault masks, are set by the currently running program to tell the processor how to respond to certain fault conditions.

## Initializing and Modifying the Arithmetic Controls

The state of the processor's arithmetic controls is undefined at processor initialization, on a processor restart (initiated with a restart processor IAC), or on a warmstart processor (initiated with a warmstart processor IAC). Part of the initialization code or restart code should thus be to set the arithmetic controls to a specific state.

The arithmetic controls can be examined and modified using the modify arithmetic controls (**modac**) instruction. This instruction uses a mask to allow specific bits to be changed.

When the processor binds itself to a process, it loads the arithmetic controls word in the process's PCB into its arithmetic controls register. When the processor suspends a process, it automatically stores the state of the arithmetic controls register in the PCB.

The processor also automatically saves and restores the arithmetic controls when it services an interrupt or handles a fault. Here, the processor saves the current state of the arithmetic controls in an interrupt record or fault record, then restores the arithmetic controls upon returning from the interrupt or fault handler, respectively.

### Functions of the Arithmetic-Controls Bits

The functions of the various arithmetic controls bits are as follows:

#### NOTE

In the following discussion, some of the arithmetic controls bits are referred to as "sticky flags."

A sticky flag is one that the processor never implicitly clears. Once the processor sets a sticky flag to indicate that a particular condition has occurred, the flag remains set until the program explicitly clears it.

### Condition-Code Flags

The processor sets the condition-code flags (bits 0-2) to indicate the results of certain instructions (usually compare instructions). Other instructions, such as conditional-branch instructions, examine these flags and perform functions according to their state. Once the processor has set these flags, it leaves them unchanged until it executes another instruction that uses these flags to store results.

These flags are used to show either true or false conditions or inequalities (greater-than, equal, or less-than conditions). Table 3-1 shows how the processor sets the flags to show true or false conditions.

**Table 3-1: Condition Codes for True or False Conditions**

Condition Code	Condition
010	true
000	false

Table 3-2 shows how the processor sets the condition-code flags to show inequalities. The term unordered is used when comparing floating-point numbers. If, when comparing two floating-point values, one of the values is a NaN (not a number), the relationship is said to be "unordered." Refer to the section in Chapter 7 titled "Comparison and Classification" for further information about the ordered and unordered conditions.

**Table 3-2: Condition Codes for Inequality Conditions**

Condition Code	Condition
000	unordered
001	greater than
010	equal
100	less than

Certain instructions (such as the branch-if instructions) use a 3-bit mask to evaluate the condition-code flags. For example the branch-if-greater-or-equal instruction (**bge**) uses a mask of  $011_2$  to determine if the condition code is set to either greater-than or equal. These masks cover the additional conditions of greater-or-equal, less-or-equal ( $110_2$ ), not-equal ( $101_2$ ), and ordered ( $111_2$ ).

### Arithmetic-Status Flags

The processor uses the arithmetic-status field (bits 3-6) in conjunction with the classify instructions (**classr** and **classrl**) to show the class of a floating-point number. When executing these instructions, the processor sets the bits in the arithmetic-status field as shown in Table 3-3, according to the class of the value being classified. The "s" bit in Table 3-3 is set to the sign of the value being classified.

**Table 3-3: Encoding of Arithmetic-Status Field**

Arithmetic Status	Classification
s000	zero
s001	denormalized number
s010	normal finite number
s011	infinity
s100	quiet NaN
s101	signaling NaN
s110	reserved operand

The remainder real instructions (**remr** and **remrl**) also use the arithmetic-status field as described in Chapter 17.

### Integer-Overflow Flag and Mask

The integer-overflow flag (bit 8) and the integer-overflow mask (bit 12) are used in conjunction with the arithmetic integer-overflow fault. The mask bit masks the integer-overflow fault. When the fault is masked, the processor sets the integer-overflow flag whenever integer overflow occurs, to indicate that the fault condition has occurred even though the fault has been masked. If the fault is not masked, the fault is allowed to occur and the flag is not set. The

integer-overflow flag is a sticky flag. (Refer to the discussion of the arithmetic integer-overflow fault in Chapter 12 for more information about the integer-overflow mask and flag.)

### No-Imprecise-Faults Flag

The no-imprecise-faults flag (bit 15) determines whether or not imprecise faults are allowed to be raised. If set, faults are required to be precise; if clear, certain faults can be imprecise. (Refer to the section in Chapter 12 titled "Precise and Imprecise Faults" for more information about this flag.)

### Floating-Point Flags and Masks

The floating-point flags (bits 16 through 20) and masks (bits 24 through 28) perform the same functions as the integer-overflow flag and mask, except they are used for operations on real (floating point) numbers. When a mask is set, its associated floating-point fault is masked. When a mask is clear, the processor sets the flag for the associated fault whenever the fault condition occurs, but does not generate a fault. All the floating-point flags are sticky bits. Refer to the section in Chapter 7 titled "Exceptions and Fault Handling" for a detailed discussion of the floating-point faults and their associated flags and masks in the arithmetic controls.

### Floating-Point-Normalizing-Mode Flag

The floating-point-normalizing-mode flag (bit 29) determines whether or not floating-point instructions are allowed to operate on denormalized numbers. If set, floating-point instructions are allowed to operate on denormalized numbers; if clear, the processor generates a floating reserved-operand fault when it detects denormalized numbers that are used as operands for floating-point instructions. (Refer to the section in Chapter 7 titled "Normalizing Mode" for more information on the use of this flag.)

### Floating-Point-Rounding Control

The floating-point-rounding-control field (bits 30-31) indicates which rounding mode is in effect for floating point computations. These bits are set as shown in Table 3-4, depending on the rounding mode to be selected.

Table 3-4: Encoding of Floating-Point-Rounding-Control Field

Rounding Control	Rounding Mode
00	round to nearest (even)
01	Round down (toward negative infinity)
10	Round up (toward positive infinity)
11	Truncate (round toward zero)

(Refer to the section in Chapter 7 titled "Rounding Control" for more information on the use of the floating-point-rounding-control field.)

All the unused bits in the arithmetic controls are reserved and must be set to 0.

## PROCESS AND TRACE CONTROLS

The processor's process controls and trace controls are also cached on the processor chip. The process controls are a set of 32 bits that control or show the status of the currently running process. The process controls are described in detail in Chapter 13. The trace controls are a set of 32 bits that control the tracing facilities of the processor. The trace controls are described in Chapter 16.

## PARTITIONING THE ADDRESS SPACE

Instructions, data, or stacks can be located anywhere in the address space, with the following exceptions. Instructions must be aligned on word boundaries. When handling a 32-bit instruction pointer, the processor generally assumes that the 2 least-significant bits of the address are zero.

The processor's virtual-memory management system requires that the address space be divided into four regions, as shown in Figure 3-4.

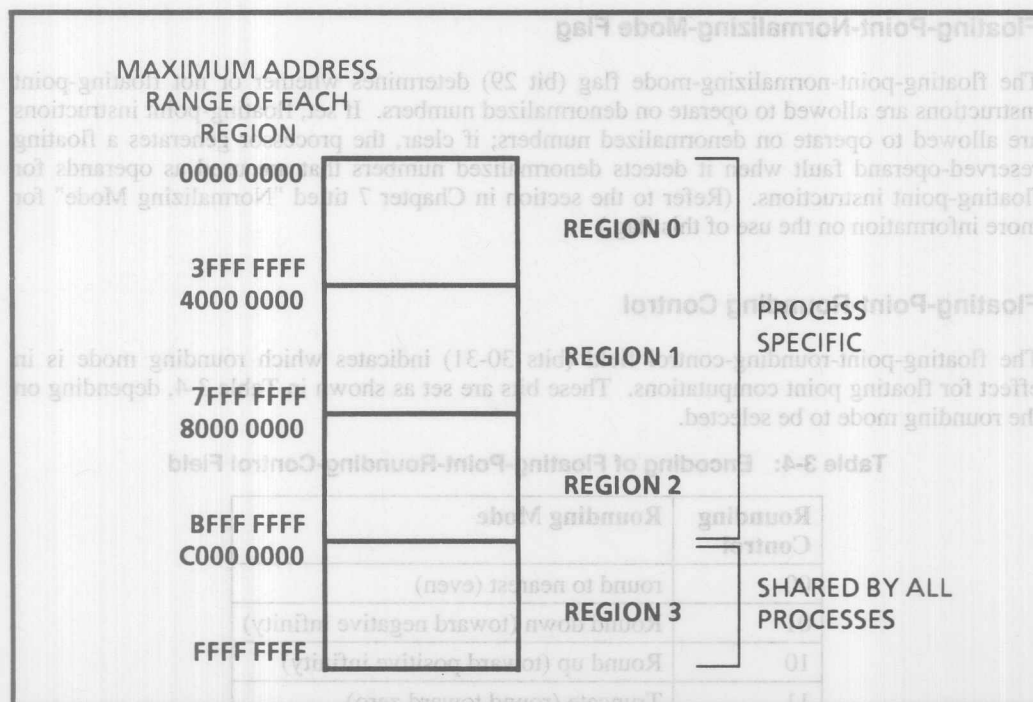


Figure 3-4: Address Space Regions



Each of these regions is managed with a separate page table or set of page tables. This allows the read and write rights of a region to be assigned on a page-by-page basis.

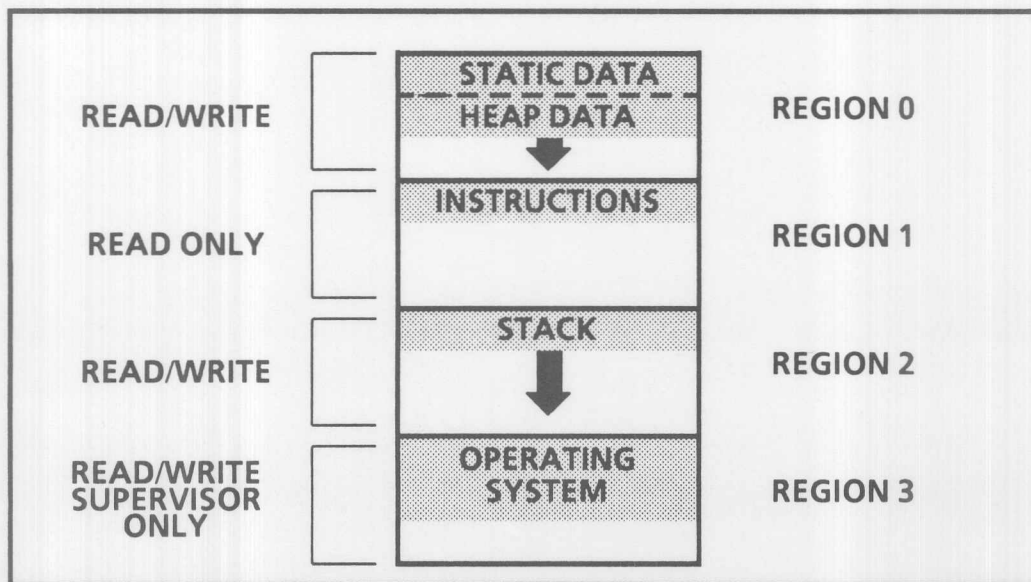
In addition, region 3 is defined to be processor specific, meaning that it is shared by all the processes that are running on the processor.

#### NOTE

Dividing the address space into regions and pages is a memory management convention that does not affect the processor's view of the address space. The processor still views the address space as being flat, with one exception. When an operand spans across one of the region boundaries shown in Figure 3-4, the results are unpredictable. This exception should be of only minor concern. However, if it does cause a problem, the section in Chapter 8 titled "Making Region Boundaries Transparent" describes how to overcome this limitation by mapping regions 0, 1, and 2 into a single page-table directory.

In the physical-addressing mode, there is no paging of the address space; so, the restriction on operands crossing region boundaries does not apply.

Figure 3-5 shows one way that the regions of the address space can be used. Here the process specific regions, regions 0, 1, and 2, are used to store the data, instructions, and procedure stack, respectively. Region 3, which all the processes share, contains system code and data, and the interrupt stack.



**Figure 3-5: Typical Use of Address-Space Regions**

This partitioning of the address space provides two benefits. First, the region containing code can be write protected. Second, the system area will not have to be swapped in and out each time there is a process switch, which reduces process switching time.

## INSTRUCTION CACHING

The processor provides a 512-byte cache for instructions. When the processor fetches an instruction or group of instructions from memory, they are stored in this cache before being fed into the instruction-execution pipeline. The processor manages this cache transparently from the program being run.

### NOTE

This instruction cache is a read-only cache, meaning that once bytes from the instruction stream are written into the instruction cache, they cannot be changed. Because of this, the processor does not support self-modified programs in a transparent fashion. The only way to change the instruction stream once it has been written into the instruction cache is to purge the instruction cache. The IAC message "purge instruction cache" is provided for this purpose, as described in Chapter 11.

Figure 3-5 shows one way that the regions of the address space can be used. Here the process specific regions, regions 0, 1, and 2, are used to store the data, instructions, and procedure stack, respectively. Region 3, which all the processes share, contains system code and data, and the interrupt stack.

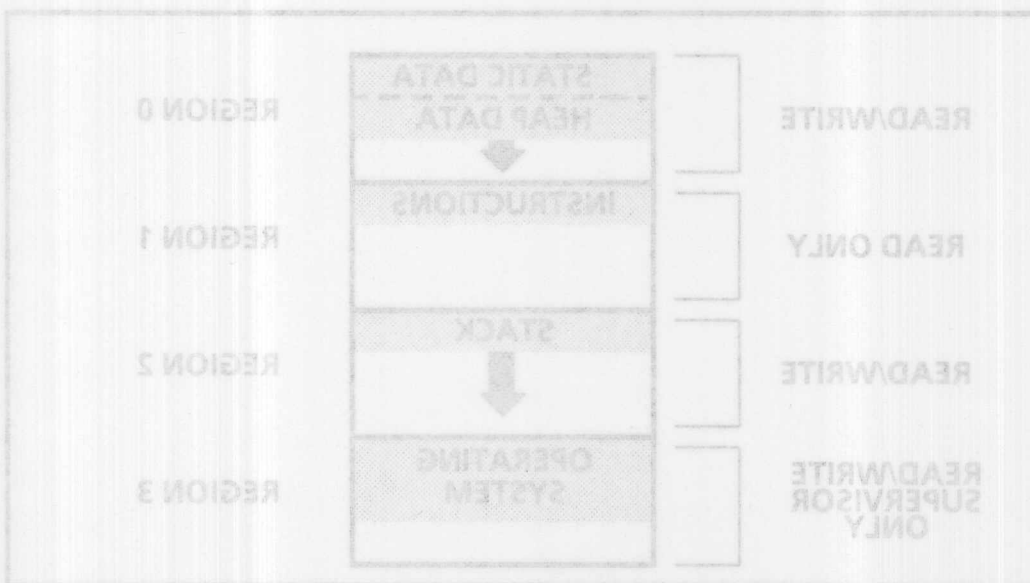


Figure 3-5: Typical Use of Address-Space Regions

This partitioning of the address space provides two benefits. First, the region containing code can be write protected. Second, the system area will not have to be swapped in and out each time there is a process switch, which reduces process switching time.

---

# *Procedure Calls*

**4**

---

---

# Procedure Calls

4

---

## CHAPTER 4 PROCEDURE CALLS

This chapter describes the 80960MC processor's procedure call and stack mechanism. It also describes the user-supervisor protection model, which provides protection for privileged procedures such as operating-system procedures.

### TYPES OF PROCEDURE CALLS

The processor supports three types of procedure calls:

- Local call
- System call
- Branch and link

A local call uses the processor's call/return mechanism, in which a new set of local registers and a new frame on the stack are allocated for the called procedure. A system call is similar to a local call, except that it provides access to procedures through a procedure table. The most important use of a system call is to call privileged procedures, called *supervisor procedures*. A system call to a supervisor procedure is called a *supervisor call*. A branch and link is merely a branch to a new instruction with the return IP stored in a global register.

In this chapter, the call/return mechanism is introduced first and is followed by a discussion of how this mechanism is used to make local calls and system calls.

#### NOTE

The processor's interrupt- and fault-handling mechanisms use implicit procedure calls. Implicit calls to interrupt-handler and fault-handler procedures are described in detail in Chapters 10 and 12, respectively.

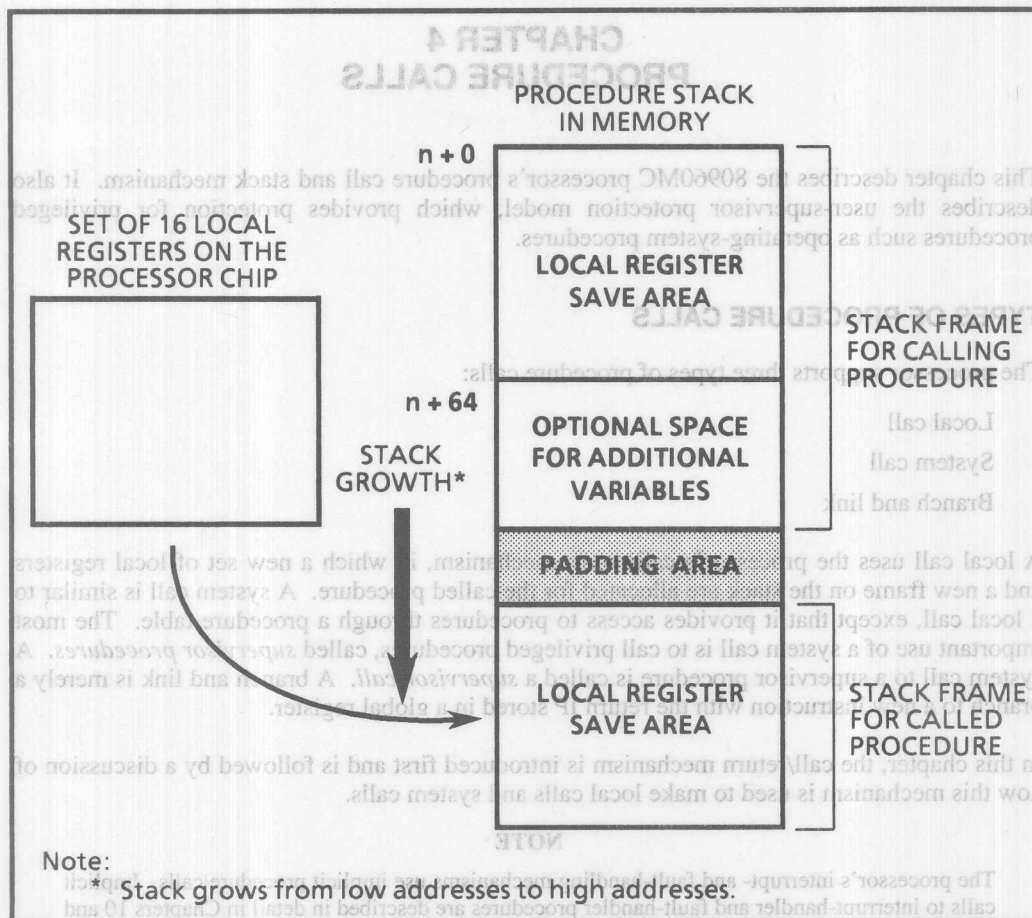
### CALL/RETURN MECHANISM

The processor's call/return mechanism has been designed to simplify procedure calls and to provide a flexible method for storing and handling variables that are local to a procedure.

Two structures support this mechanism: the local registers (on the processor chip) and the procedure stack (in memory). Figure 4-1 shows the relationship of the local registers to the procedure stack. For each procedure, the processor automatically allocates a set of local registers and a frame on the procedure stack. Since the local registers are on-chip, they provide fast-access storage for local variables. If additional space for local variables is required, it can be allocated in the stack frame.

When a procedure call is made, the processor automatically saves the contents of the local registers and the stack frame for the calling procedure and sets up a new set of local registers and a new stack frame for the called procedure.





**Figure 4-1: Local Registers and Procedure Stack**

This procedure-call mechanism provides two benefits. First, it provides a structure for storing a virtually unlimited number of local variables for each procedure: the on-chip local registers provide quick access to often-used variables and the stack provides space for additional variables.

Second, a program does not have to explicitly save and restore the variables stored in the local registers and stack frames. The processor does this implicitly on procedure calls and on returns.

A detailed description of the call/return mechanism is given in the following paragraphs.

## Local Registers and the Procedure Stack

For each procedure, the processor allocates a set of 16 local registers. Three of these registers (r0, r1, and r2) are reserved for linkage information to tie procedures together. The remaining 13 local registers are available for general storage of variables.

For each process, the processor maintains a procedure stack in memory. This stack can be located anywhere in the address space and grows from low addresses to high addresses.

The stack consists of contiguous frames, one frame for each active procedure. As shown in Figure 4-2, each stack frame provides a save area for the local registers and an optional area for additional variables.

To increase the speed of procedure calls, the 80960MC processor provides four sets of local registers. Thus, when a procedure call is made, the contents of the current set of local registers often do not have to be stored in the procedure stack. Instead, a new set of local registers is assigned to the called procedure. When the number of nested procedure calls exceeds the number of register sets, the processor automatically stores the contents of the oldest set of local registers on the stack to free up a set of local registers for the most recently called procedure.

Refer to the section later in this chapter titled "Mapping the Local Registers to the Procedure Stack" for further discussion of the relationship between the local-register sets and the procedure stack.

## Procedure-Linking Information

Global register g15 (FP) and local registers r0 (PFP), r1 (SP), and r2 (RIP) contain information to link procedures together and to link the local registers to the procedure stack. The following paragraphs describe this linkage information.

### Frame Pointer

The FP is the address of the first byte of the current (topmost) stack frame. It is stored in global register g15. The 80960MC processor aligns each new stack frame on a 64-byte boundary. Since the resulting FP always points to a 64-byte boundary, the processor ignores the 6 low-order bits of the FP and interprets them to be zero.

### Stack Pointer

The SP is the address of the next available byte of the stack frame, which can also be thought of as the last byte of the stack frame plus one. It is stored in local register r1. The procedure stack grows upward (i.e., toward higher addresses). To determine the initial SP value, the processor adds 64 to the FP.

Figure 4-2: Procedure Stack Structure

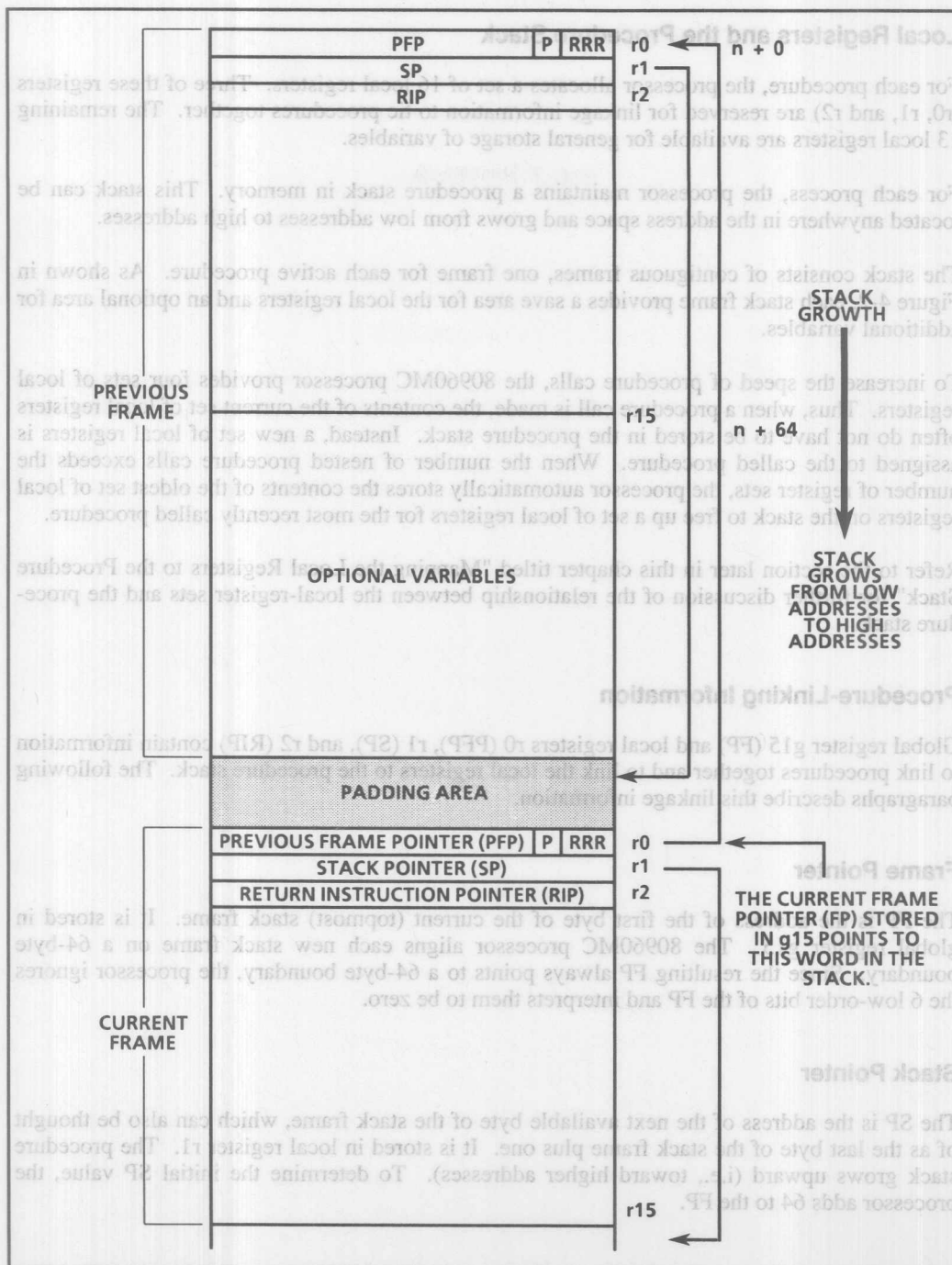


Figure 4-2: Procedure Stack Structure

If additional space is needed on the stack for local variables, the SP may be incremented in one-byte increments. For example, the following instruction adds six words of additional space to the stack:

```
addo sp, 24, sp # sp ← sp + 24
```

With the Intel 80960MC Assembler, the keyword "sp" stands for register r1.

#### NOTE

The SP should be incremented before additional variables are added to the stack. This practice prevents errors that might occur if data is added to the stack and a process switch occurs before the SP has been incremented.

### Padding Area

When the processor creates a new frame on a procedure call, it will, if necessary, add a padding area to the stack so that the new frame starts on a 64 byte boundary. To create the padding area, the processor rounds off the SP for the current stack frame (the value in r1) to the next highest 64 byte boundary. This value becomes the FP for the new stack frame.

### Previous-Frame Pointer

The PFP is the address of the first byte of the previous stack frame. It is stored in local register r0. Since the 80960MC ignores the 6 low-order bits of the FP, only the 26 most-significant bits of the PFP are stored here. The 4 least-significant bits of r0 are then used to store return status information.

### Return Status and Prereturn-Trace Information

Bits 0 through 2 of local register r0 contain return status information for the calling procedure and bit 3 contains the prereturn-trace flag. When a procedure call is made (either explicit or implicit), the processor records the call type in the return status field. The processor then uses this information to select the proper return mechanism when returning to the calling procedure.

Table 4-1 shows the encoding of the return status field according to the different types of calls that the processor supports. Of the five types of calls allowed, the fault call (described in Chapter 12) and the interrupt and idle-interrupt calls (described in Chapter 10) are implicit calls that the processor initiates. The local call (described in this section) is an explicit call that a program initiates using the **call** or **callx** instruction. The supervisor call (described at the end of this chapter in the section titled "System Call/Return Mechanism") is an explicit call that a program makes using the **calls** instruction.

The third column of Table 4-1 shows the type of a return action that the processor takes depending on the state of the return status field.

The processor records two versions of the supervisor call: one for when the trace-enable flag in the process controls is set prior to a supervisor call and one for when the flag is clear prior to the call. The trace controls are described in detail in Chapter 16.

**Table 4-1: Encoding of Return-Status Field**

Encoding	Call Type	Return Action
000	Local call or supervisor call made from the supervisor mode	Local return
001	Fault call	Fault return
010	Supervisor call from user mode, trace was disabled before call	Supervisor return, with the trace enable flag in the process controls set to 0 and the execution mode flag set to 0
011	Supervisor call from user mode, trace was enabled before call	Supervisor return, with the trace enable flag in the process controls set to 1 and the execution mode flag set to 0
100	reserved	
101	reserved	
110	Idle-interrupt call	Idle-interrupt return
111	Interrupt call	Interrupt return

The prereturn-trace flag is used in conjunction with the call-trace and prereturn-trace modes. If the call-trace mode is enabled when a call is made, the processor sets the prereturn-trace flag; otherwise it clears the flag. Then, if this flag is set and the prereturn-trace mode is enabled, a prereturn trace event is generated on a return before any actions associated with the return operation are performed. Refer to Chapter 16 for a detailed discussion of the interaction of the call-trace and prereturn-trace modes and the prereturn-trace flag.

### Return-Instruction Pointer

The RIP is the address of the instruction that the processor is to execute after returning from a procedure call. It is stored in local register r2. When the processor executes a procedure call it sets the RIP to the address of the instruction immediately following the procedure call instruction. (Refer to the section later in this chapter titled "Local-Call Operation" for further information the RIP.)

Since the processor uses the same procedure call mechanism to make implicit procedure calls to service faults and interrupts, programs should not use register r2 for purposes other than to hold the RIP.

When a process is suspended, the IP of the next instruction is stored in r2 of the current set of local registers. When the process resumes, the processor sets the IP to the value in this register.



The availability of multiple register sets cached on the processor chip and the saving and restoring of these register sets in stack frames should be transparent to most programs. However, the following additional information about how the local registers and procedure stack are mapped to one another can help avoid problems.

Since the local-register sets reside on the processor chip, the processor will often not have to access the stack frame in the procedure stack, even though space has been allocated on the stack for the current frame. The processor only accesses the current frame in the procedure stack in the following instances:

1. to read or write variables other than those held in the local registers,
2. to read local registers that were stored in the procedure stack when the number of nested procedures calls exceeded the number of local registers, or
3. to read local registers that were stored in the procedure stack due to the suspension of the process.

This method of mapping the local registers to the register-save areas in the procedure stack has several implications. First, storing information in a local register does not guarantee that it will be stored in its associated word in the current stack frame. Likewise, storing information in the first 16 words of a stack frame does not guarantee that the local registers associated with the stack frame are modified.

Second, if you try to read the contents of the current set of local registers through a memory access to the first 16 words of the current stack frame, you may not get the expected result. This is also true if you try to read the contents of a previously stored set of local registers through a memory address to its associated stack frame.

The processor automatically stores the contents of a local register set into the register-save area of its associated stack frame only in the following two circumstances:

1. if procedure calls (local or supervisor) are nested deeper than the number of local-register sets, or
2. if the process is suspended.

Occasionally, it is necessary to have the contents of all local-register sets match the contents of the register-save areas in their associated stack frames. For example, when debugging software it may be necessary to trace the call history back through the nested procedures. This can not be done unless the cached local-register frames are flushed (i.e., written out to the procedure stack).

The processor provides two methods of voluntarily flushing the local registers: the **flushreg** (flush local registers) instruction and the flush-local-registers IAC. Both the **flushreg** instruction and the flush-local-registers IAC cause the contents of all the local-register sets, except the current set, to be written to their associated stack frames in memory.

Third, if you need to modify the PFP in register r0, you should precede this operation with the **flushreg** instruction, or else the behavior of the **ret** (return) instruction is not predictable.

Fourth, local registers should not be used for passing parameters between procedures. (Parameter passing is discussed in the following section.)

Fifth, when a set of local registers is assigned to a new procedure, the processor may not clear or initialize these registers. The initial contents of these registers are therefore unpredictable. Also, the processor does not initialize the local register-save area in the newly created stack frame for the procedure, so its contents are equally unpredictable.

## LOCAL CALL

A local call is made using either of two local call instructions: **call** and **callx**. These instructions initiate a procedure call using the call/return mechanism described earlier in this chapter.

The **call** instruction specifies the address of the called procedure as the IP plus a signed, 24-bit displacement (i.e.,  $-2^{23}$  to  $2^{23} - 4$ ).

The **callx** instruction allows any of the addressing modes to be used to specify the procedure address. The *IP with displacement* addressing mode allows full 32-bit IP relative addressing.

The **ret** instruction initiates a procedure switch back to the last procedure that issued a call.

## Local-Call Operation

During a local call, the processor performs the following operations:

1. Stores the RIP in current local-register r2.
2. Allocates a new set of local registers for the called procedure.
3. Allocates a new frame on the procedure stack.
4. Changes the instruction pointer to point to the first instruction in the called procedure.
5. Stores the FP for the calling procedure in new local-register r0 (PFP).
6. Stores the FP for the new frame in global register g15.
7. Allocates a save area for the new local registers in the new stack frame.
8. Stores the SP in new local-register r1.

## Local-Return Operation

On a return, the processor performs these operations:

1. Sets the FP in global register g15 to the value of the PFP in current local-register r0.
2. Deallocates the current local registers for the procedure that initiated the return and switches to the local registers assigned to the procedure being returned to.
3. Deallocates the stack frame for the procedure that initiated the return.
4. Sets the IP to the value of the RIP in new local-register r2.

The algorithms that the **call**, **callx**, and **ret** instructions use are described in greater detail in Chapter 17.

## PARAMETER PASSING

The processor supports two mechanisms for passing parameters between procedures: global registers and argument list.

### Passing Parameters in Global Registers

The global registers provide the fastest method of passing parameters. Here, the calling procedure copies the parameters to be passed into global registers. The called procedure then copies the parameters (if necessary) out of the global registers after the call.

On a return, the called procedure can copy result parameters into global registers prior to the return, with the calling procedure copying them out of the global registers after the return.

### Passing Parameters in an Argument List

When more parameters need to be passed than will fit in the global registers, they can be placed in an argument list. This argument list can be stored anywhere in memory providing that the procedure being called has a pointer to the list. Commonly, a pointer to the argument list is placed in a global register.

Parameters can also be returned to the calling procedure through an argument list. Here again, a pointer to the argument is generally returned to the calling procedure through a global register.

The argument list method of passing parameters should be thought of as an escape mechanism and used only when there are not enough global registers available for passing parameters.

### Passing Parameters Through the Stack

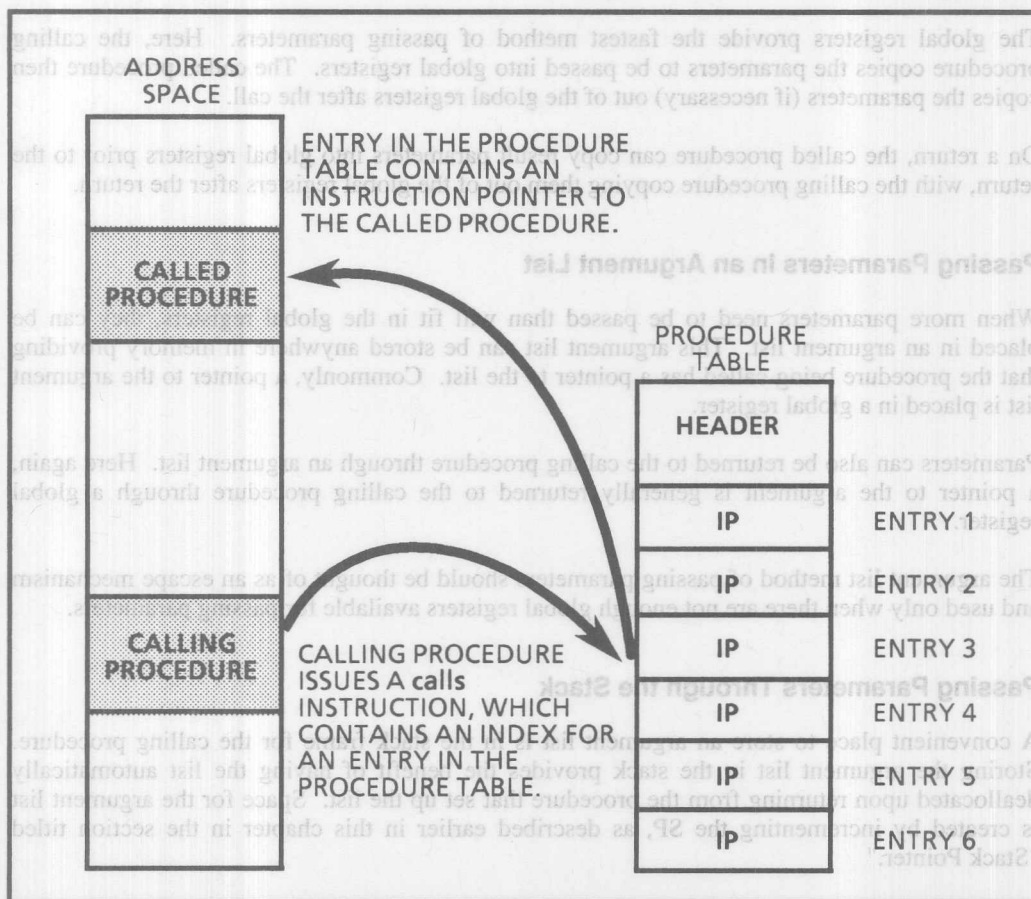
A convenient place to store an argument list is in the stack frame for the calling procedure. Storing the argument list in the stack provides the benefit of having the list automatically deallocated upon returning from the procedure that set up the list. Space for the argument list is created by incrementing the SP, as described earlier in this chapter in the section titled "Stack Pointer."

Parameters can also be returned to the calling procedure through an argument list in the stack. However, care should be taken when doing this. The return argument list must not be placed in the frame for the called procedure, since this frame is deallocated on the return. Also, if the return list is to be placed in the frame of the calling procedure, the calling procedure must allocate space for this list prior to making the call.

## SYSTEM CALL

A system call is made using the call system instruction **calls**. This call is similar to a local call except that the processor gets the IP for the called procedure from a data structure called the procedure table. (System calls are sometimes referred to in this manual as "system-procedure-table calls.")

Figure 4-3 illustrates the use of the procedure table in a system call. The **calls** instruction requires a procedure-number operand. This procedure number provides an index into the procedure table, which contains IPs for specific procedures.



**Figure 4-3: System-Call Mechanism**

The system-call mechanism supports two types of procedure calls: local calls and supervisor calls. A local call is the same as that made with the **call** and **callx** instructions, except that the processor gets the IP of the called procedure from a procedure table. The supervisor call differs from the local call in two ways: (1) it causes the processor to switch to another stack (called the supervisor stack), and (2) it causes the processor to switch to a different execution mode.

The system-call mechanism offers two benefits. First, it supports portability for application software. System calls are commonly used to call kernel services. By calling these services with a procedure number rather than a specific IP, applications software does not have to be changed each time the implementation of the kernel services is modified.

Second, the ability to switch to a different execution mode and stack allows kernel procedures and data to be insulated from applications code. This benefit is described in more detail later in this chapter in the section titled "User-Supervisor Protection Model".

## PROCEDURE TABLE

The procedure table is a general structure, which the processor uses in two ways. The first way is as a place for storing IPs for kernel procedures, which can then be accessed through the system-call mechanism described above. Here, the procedure table is called the *system-procedure table*. The processor gets a pointer to the system-procedure table from the processor control block (PRCB) as described in Chapter 9 in the section titled "System Data-Structure Pointers."

The second way a procedure table is used is as a place for storing IPs for fault-handler procedures. Here, the processor gets a pointer to the procedure table from entries in the fault table, as described in Chapter 12 in the section titled "Fault-Table Entries."

The structure of the procedure table is shown in Figure 4-4. It is 1088 bytes in length and can have up to 260 procedure entries. The following sections describe the fields in a procedure table.

### Procedure Entries

The procedure entries specify the target IPs for the procedures that can be accessed through the procedure table. Each entry is one word in length and is made up of an address (or IP) field and a type field. The address field gives the address of the first instruction of the target procedure. Since all instructions are word aligned, only the 30 most-significant bits of the address are given. The processor automatically provides zeros for the least-significant bits. Entry 0 begins at byte 48 of the procedure table; the table can contain up to 260 entries.

The procedure entry type field indicates the type of call to execute: local or supervisor. The encodings of this field are shown in Table 4-2.

Table 4-2: Encodings of Entry Type Field in Procedure Table Entry

Entry Type Field	Procedure Type
00	local procedure
01	reserved
10	supervisor procedure
11	reserved



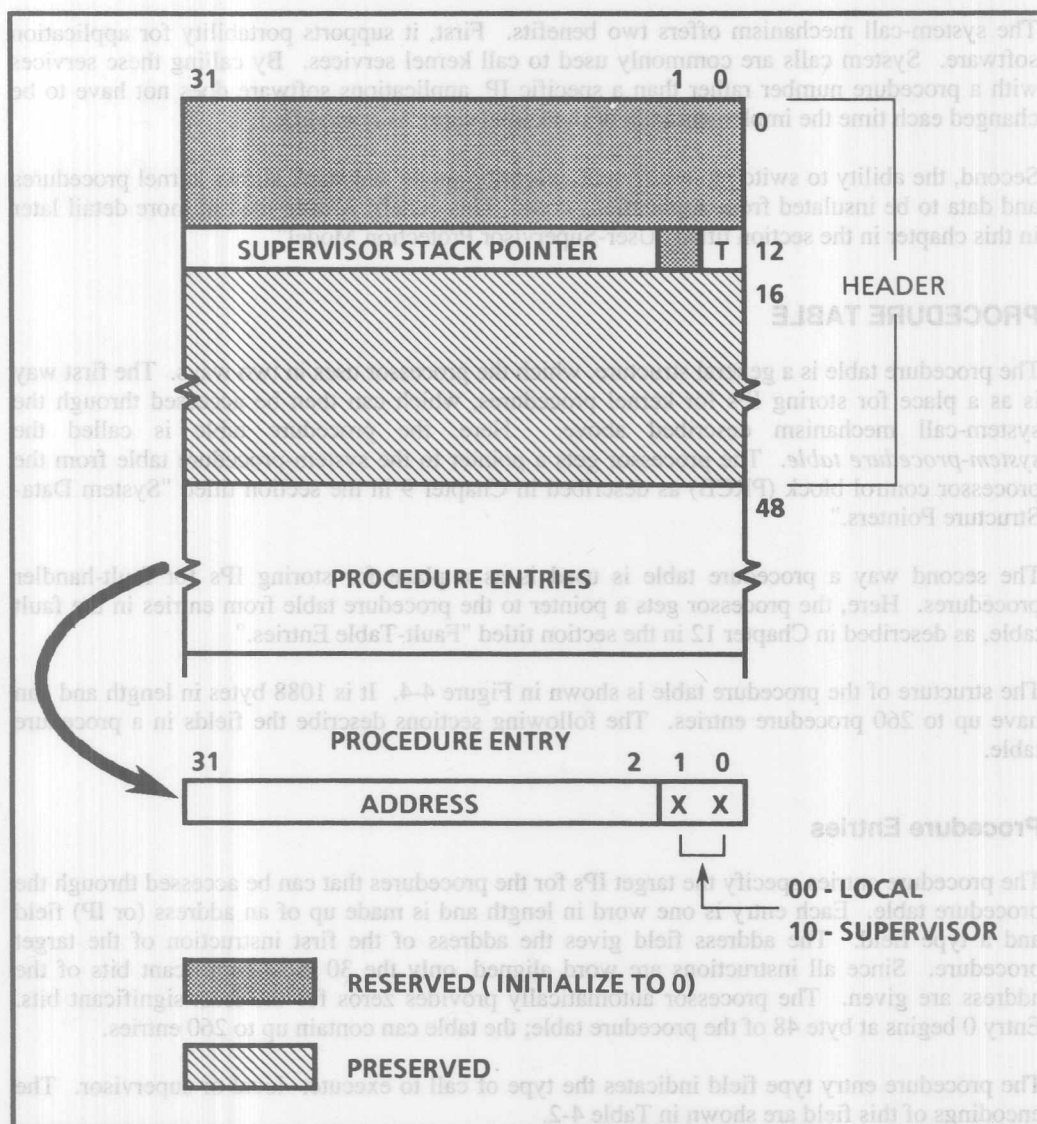


Figure 4-4: Procedure-Table Structure

### Supervisor-Stack Pointer

When a supervisor call is made, the processor switches to a new stack called the *supervisor stack*. The processor gets a pointer to this stack from the supervisor-stack-pointer entry (bytes 12-15, bits 2-31) in the procedure table. Only the 30 most-significant bits of the supervisor-stack pointer are given. The processor aligns this value to the next 64-byte boundary to determine the first byte of the new stack frame.

## Trace-Control Flag

The trace-control flag (byte 12, bit 0) specifies the new value of the trace-enable flag of the process when a supervisor call causes a switch from user mode to supervisor mode. Setting this flag to 1 enables tracing; setting it to 0 disables tracing. The use of this flag is described in the section in Chapter 16 titled "Trace Control on Supervisor Calls."

## System Call to a Local Procedure

When a **calls** instruction references a procedure entry designated as a local type (00<sub>2</sub>), the processor executes a local call to the procedure selected from the system procedure table. Neither a mode switch nor a stack switch occurs.

The **ret** instruction permits returns from either a local procedure or a supervisor procedure. The return status field in local register r0 determines the type of return action that the processor is to take. If the return status field is set to 000<sub>2</sub>, a local return is executed. In a local return, no stack or mode switching is carried out.

## USER-SUPERVISOR PROTECTION MODEL

The processor provides a two-state protection model called the user-supervisor protection model. With this model, access to selected procedures and data structures can be restricted by means of page protection and mode switching between two execution modes: user and supervisor.

This protection model allows a system to be designed in which kernel code and data reside in the same address space as user code and data, but access to the kernel procedures (called supervisor procedures) is only allowed through a procedure table, which forms a tightly controlled and protected interface. This interface is provided by the system procedure table.

The user-supervisor protection model also allows kernel procedures to be executed using a different stack (the supervisor stack) than is used to execute applications program procedures. The ability to switch stacks helps maintain the integrity of the kernel. For example, it would allow system debugging software or a system monitor to be accessed, even if an applications program crashes.

## User and Supervisor Modes

When using the user-supervisor protection model, a process can be in either of two execution modes: user or supervisor. The difference between the two modes is that when the process is in the supervisor mode, it has the following additional privileges:

- It may access pages that have supervisor-only rights. (A program cannot access these pages in the address space when the process is in the user mode.)
- It may use additional instructions. These instructions typically control process management and kernel functions.

## Supervisor Calls

Mode switching between the user and supervisor execution modes is accomplished through a supervisor call. A supervisor call is a call executed with the **calls** instruction that references a supervisor procedure in the system procedure table (i.e., a procedure with an entry type 10<sub>2</sub>).

When the processor is in the user mode and it executes a **calls** instruction, the processor performs the following actions:

- It switches to supervisor mode
- It switches to the supervisor stack
- It sets the return status field in register R0 of the calling procedure to 01X<sub>2</sub>, indicating that a mode and stack switch has occurred.

The processor remains in the supervisor mode until a return is performed from the procedure that caused the original mode switch. While in the supervisor mode, either the local call instructions (**call** and **callx**) or the **calls** instruction can be used to call supervisor procedures.

(The **call** and **callx** instructions call local (or user) procedures in user mode and supervisor procedures in supervisor mode. There is no stack or processor state switching associated with these instructions.)

When a **ret** instruction is executed and the return status field is set to 01X<sub>2</sub>, the processor performs a supervisor return. Here, the processor switches from the supervisor stack to the local stack, and the execution mode is switched from supervisor to user.

## Supervisor Stack

When using the user-supervisor mechanism, the processor maintains separate stacks in the address space, one for procedures executed in the user mode (local procedures) and another for procedures executed in the supervisor mode (supervisor procedures). When in the user mode, the local procedure stack (described at the beginning of this chapter) is used. When a supervisor call is made, the processor switches to the supervisor stack. It continues to use the supervisor stack until a return is made to the user mode.

The structure of the supervisor stack is identical to that of the local procedure stack (shown in Figure 4-2). The processor obtains the SP for the supervisor stack from the procedure table. When a supervisor call is executed while in the user mode (causing a switch to the supervisor stack), the processor aligns this SP to the next 64-byte boundary to form the new FP for the supervisor stack. When a local call or supervisor call is made while in the supervisor mode, the processor aligns the SP in the current frame of the supervisor stack to the next 64-byte boundary to form the FP pointer. This operation allows supervisor procedures to be called from supervisor procedures.

## BRANCH AND LINK

The **bal** (branch and link) and **balx** (branch and link extended) instructions provide an alternate method of making procedure calls. These instructions save the address of the next instruction (RIP) in a specified location, then branch to a target instruction or set of instructions. The state of the local registers and stack remains unchanged. (For the **bal** instruction, the RIP is automatically stored in global register g14; for the **balx** instruction, the location of the RIP is specified with one of the instruction operands.)

A return is accomplished with a **bx** (branch extended) instruction, where the address of the target instruction is the one saved with the branch and link instruction.

Branch and link procedure calls are recommended for calls to procedures that (1) do not call other procedures (i.e., for procedure calls that do not result in nesting of procedures) and (2) do not need many local variables (i.e., allocation of a new set of local registers does not provide any benefit). Here, local registers as well as global registers can be used for parameter passing.







---

# Data Types and Addressing Modes

---

5

## CHAPTER 5

# DATA TYPES AND ADDRESSING MODES

This chapter describes the data types that the 80960MC processor recognizes and the addressing modes that are available for accessing memory locations.

### DATA TYPES

The processor defines and operates on the following data types:

- Integer (8, 16, 32, and 64 bits)
- Ordinal (8, 16, 32, and 64 bits)
- Real (32, 64, and 80 bits)
- Decimal (ASCII digits)
- Bit Field
- Byte String
- Triple-Word (96 bit)
- Quad-Word (128 bit)

The integer, ordinal, real, and decimal data types can be thought of as numeric data types because some operations on these data types produce numeric results (e.g., add, subtract).

The remaining data types (bit field, byte string, triple word, and quad word) represent groupings of bits or bytes that the processor can operate on as a whole, regardless of the nature of the data contained in the group. These data types facilitate moving and operating on blocks of bits or bytes.

### Integers

Integers are signed whole numbers, which are stored and operated on in two's complement format. The processor recognizes four sizes of integers: 8 bit (byte integers), 16 bit (short integers), 32 bit (integers), and 64 bit (long integers). Figure 5-1 shows the formats for the four integer sizes and the ranges of values allowed for each size.

### Ordinals

Ordinals are a general-purpose data type. The processor recognizes four sizes of ordinals: 8 bit (byte ordinals), 16 bit (short ordinals), 32 bit (ordinals), and 64 bit (long ordinals). Figure 5-2 shows the formats for the four ordinal sizes and the ranges of numeric values allowed for each size.

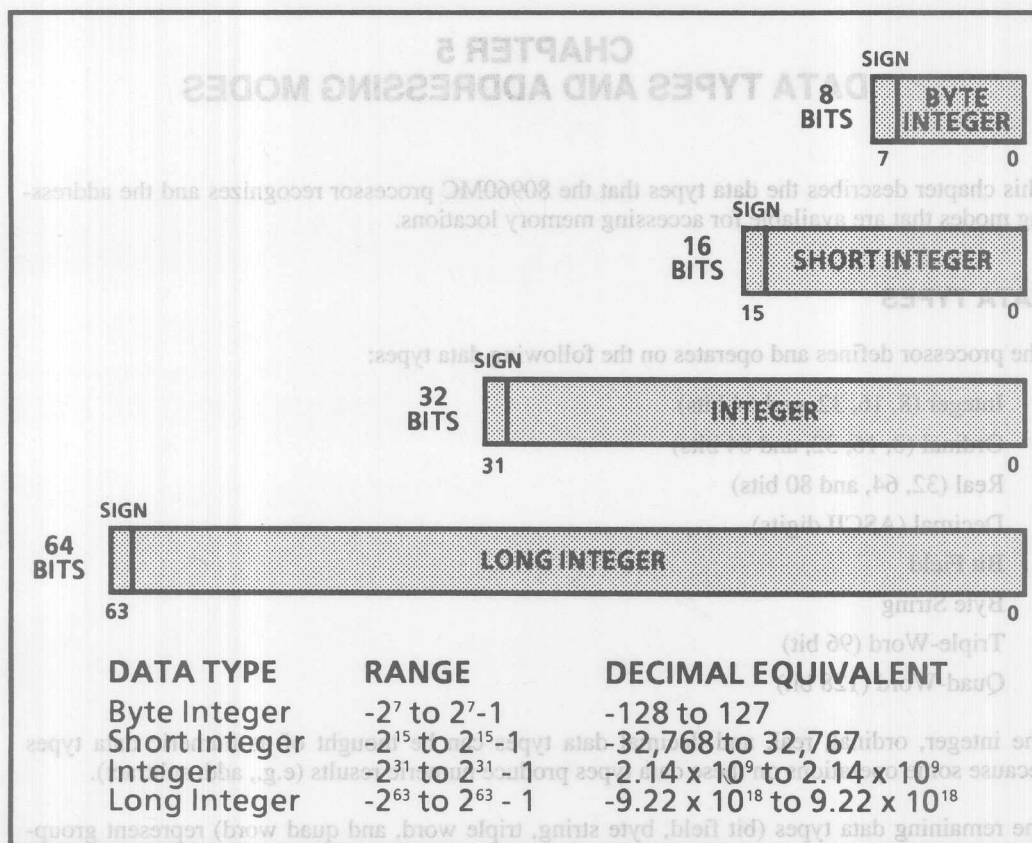


Figure 5-1: Integer Format and Range

The processor uses ordinals for both numeric and non-numeric operations. For numeric operations, ordinals are treated as unsigned whole numbers. The processor provides several arithmetic instructions that operate on ordinals. For non-numeric operations, ordinals contain bit fields, byte strings, and Boolean values.

When ordinals are used to represent Boolean values, a  $1_2$  represents a TRUE and a  $0_2$  represents a FALSE.

## Reals

Reals are floating-point numbers. The processor recognizes three sizes of reals: 32 bit (reals), 64 bit (long reals), and 80 bit (extended reals). The real-number format conforms to ANSI/IEEE Std. 754-1985, the IEEE Standard For Binary Floating-Point Arithmetic. Real numbers are discussed in greater detail in Chapter 7.

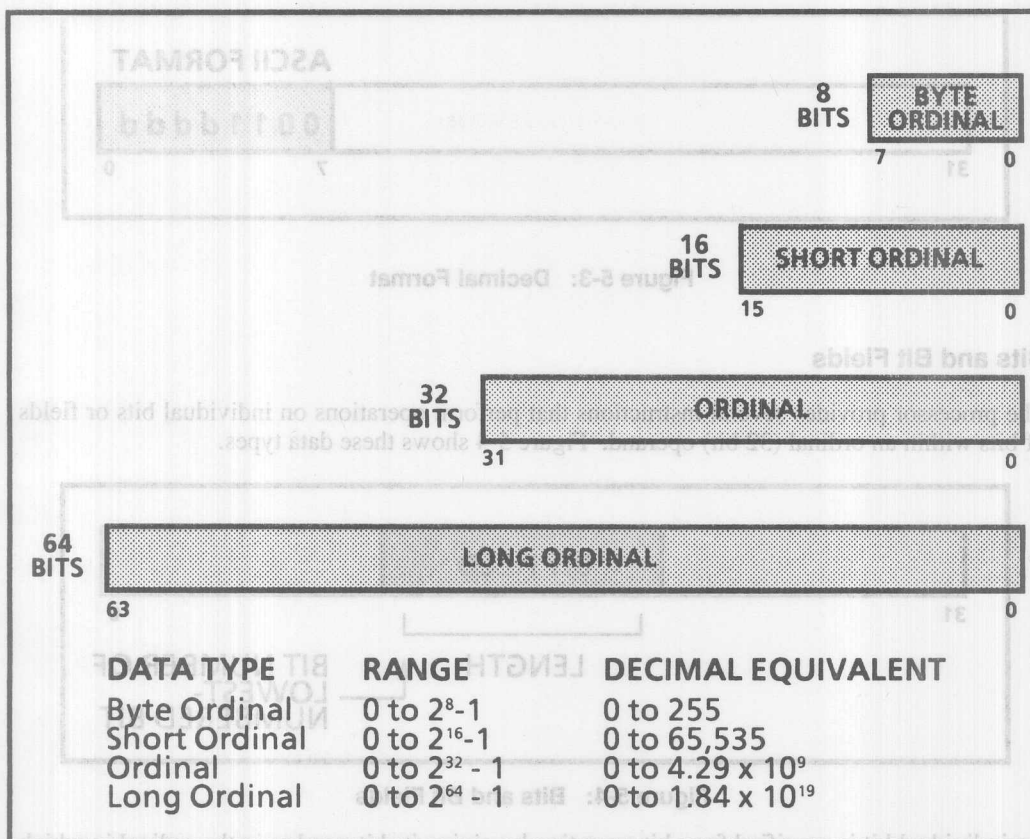


Figure 5-2: Ordinal Format and Range

## Decimals

The processor provides three instructions that perform operations on decimal values when the values are presented in ASCII format. Figure 5-3 shows the ASCII format for decimal digits. Each decimal digit is contained in the least-significant byte of an ordinal (32 bits). The decimal digit must be of the form  $0011\text{dddd}_2$ , where  $\text{dddd}_2$  is a binary-coded decimal value from 0 to 9. For decimal operations, bits 8 through 31 of the ordinal containing the decimal digit are ignored.



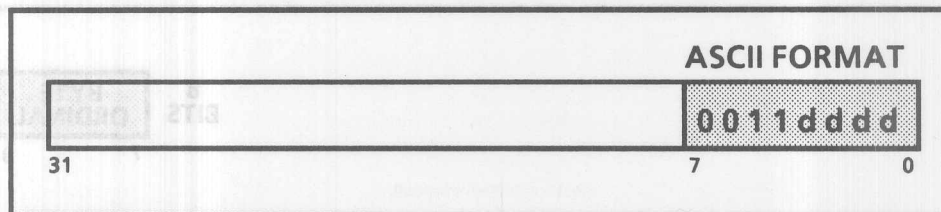


Figure 5-3: Decimal Format

### Bits and Bit Fields

The processor provides several instructions that perform operations on individual bits or fields of bits within an ordinal (32 bit) operand. Figure 5-4 shows these data types.

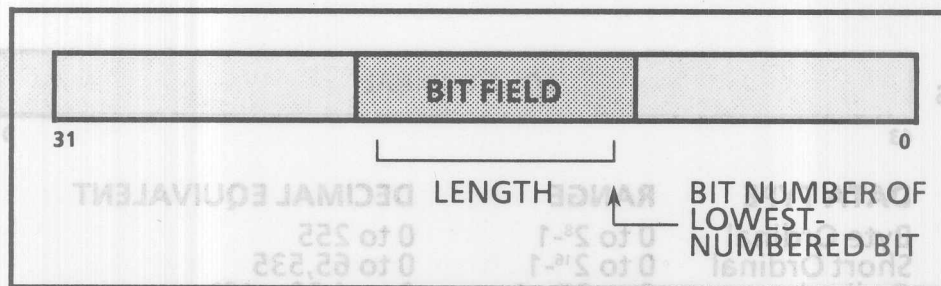


Figure 5-4: Bits and Bit Fields

An individual bit is specified for a bit operation by giving its bit number in the ordinal in which it resides. The least-significant bit of a 32-bit ordinal is bit 0; the most-significant bit is bit 31.

A bit field is a contiguous sequence of bits of from 0 to 32 bits in length within a 32-bit ordinal. A bit field is defined by giving its length in bits and the bit number of its lowest-numbered bit.

A bit field cannot span a register boundary.

### Byte String

A byte string is a contiguous sequence of byte ordinals. The length of a byte string is the number of bytes in the string; a length of zero specifies an empty string. The maximum length of a byte string is  $2^{32} - 1$  bytes.

Byte-string operations are performed on byte strings in memory. The address of a byte string is the address of the first byte in the string. Consecutive bytes of the string are stored in increasing byte addresses.

## Triple and Quad Words

Triple and quad words refer to consecutive bytes in memory or in registers: a triple word is 12 bytes and a quad word is 16 bytes. These data types facilitate the moving of blocks of bytes. The triple-word data type is useful for moving extended-real numbers (80 bits).

The quad-word instructions (**ldq**, **stq**, and **movq**) offer the most efficient way to move large blocks of data.

## BYTE, WORD, AND BIT ADDRESSING

The processor provides instructions for moving blocks of data values of various lengths from memory to registers (load) and from registers to memory (store). The allowable sizes for blocks are bytes, half-words (2 bytes), words (4 bytes), double words, triple words, and quad words. For example, the **stl** (store long) instruction stores an 8-byte (double word) block of data in memory.

When a block of data is stored in memory, the least-significant byte of the block is stored at a base memory address and the more significant bytes are stored at successively higher addresses.

When loading a byte, half-word, or word from memory to a register, the least-significant bit of the block is always loaded in bit 0 of the register. When loading double words, triple words, and quad words, the least-significant word is stored in the base register. The more significant words are then stored at successively higher numbered registers. Double words, triple words, and quad words must also be aligned in registers to natural boundaries as described in Chapter 3 in the section titled "Register Alignment."

Bits can only be addressed in data that resides in a register. Bit 0 in a register is the least-significant bit and bit 31 is the most-significant bit.

## LITERALS

The processor recognizes two types of literals (ordinal literals and floating-point literals), which can be used as operands in some instructions. An ordinal literal can range from 0 to 31 (5 bits). When an ordinal literal is used as an operand, the processor expands it to 32 bits by adding leading zeros. If the instruction defines an operand larger than 32 bits, the processor zero-extends the value to the operand size. If an ordinal literal is used in an instruction that requires integer operands, the processor treats the literal as a positive integer value.

The processor also recognizes two floating-point literals (+0.0 and +1.0). These floating-point literals can only be used with floating-point instructions. As with the ordinal literals, the processor converts the floating-point literals to the operand size specified by the instruction.

### NOTE

A few of the floating-point instructions use both floating-point and non-floating-point operands (e.g., the convert integer-to-real instructions). Ordinal literals can be used in these instructions for non-floating-point operands.

## REGISTER ADDRESSING

A register may be used as an operand in an instruction by giving the register's number (e.g., g0, r5, fp3). Both floating-point and non-floating-point instructions can reference global and local registers in this way. However, floating-point registers can only be referenced in conjunction with a floating-point instruction.

## MEMORY-ADDRESSING MODES

The processor offers 9 modes for addressing operands in memory. These modes are grouped as follows:

- Absolute
- Register Indirect
- Register Indirect with Index
- Index with Displacement
- IP with Displacement

Each addressing mode is used to reference a byte address in the processor's address space. Table 5-1 shows all the memory-addressing modes, a brief description of the elements of the address in each mode, and the assembly-code syntax for each mode.

**Table 5-1: Addressing Modes**

Mode	Description	Assembler Syntax
Absolute offset	offset	exp
Register Indirect	abase	(reg)
Register Indirect with offset	abase + offset	exp (reg)
Register Indirect with index	abase + (index*scale)	(reg) [reg*scale]
Register Indirect with index and displacement	abase + (index*scale) + displacement	exp (reg) [reg*scale]
Index with displacement	(index*scale) + displacement	exp [reg*scale]
IP with displacement	IP + displacement + 8	exp (IP)

Where: *reg* is register and *exp* is expression

### NOTE

A few of the floating-point instructions use both floating-point and non-floating-point operands (e.g., the convert integer-to-real instructions). Original literals can be used in these instructions for non-floating-point operands.

### **Absolute**

Absolute addressing is used to reference a memory location directly as an offset from address 0 of the address space, ranging from  $-2^{31}$  to  $2^{31} - 1$ . Typically, an assembler will allow absolute addresses to be specified through arithmetic expressions (e.g.,  $x + 44$ ), symbolic labels, and absolute values.

At the machine-level, two absolute-addressing modes are provided, depending on the instruction format (i.e., MEMA or MEMB). For the MEMA format, the offset is an ordinal number ranging from 0 to 2047; for the MEMB format, the offset is an integer (called a displacement) ranging from  $-2^{31}$  to  $2^{31} - 1$ . After evaluating an absolute address, the assembler will convert the address into an offset and select the appropriate machine-level instruction type and addressing mode. (The machine-level addressing modes and instruction formats are described in Appendix B.)

### **Register Indirect**

The register indirect addressing modes allow an address to be specified with an ordinal value (32 bits) in a register or with an offset or a displacement added to a value in a register. Here, the value in the register is referred to as the address base (abase).

Again, an assembler will allow the offset and displacement to be specified with an expression or symbolic label, then evaluate the address to determine whether an offset or a displacement is appropriate.

### **Register Indirect with Index**

The register indirect with index addressing modes allow a scaled index to be added to the value in a register. The index is specified by means of a value placed in a register. This index value is then multiplied by the scale factor. The allowable scale factors are 1, 2, 4, 8, and 16.

A displacement may also be added to the abase value and scaled index.

### **Index with Displacement**

A scaled index can also be used with a displacement alone. Again, the index is contained in a register and is multiplied by a scaling constant before the displacement is added to it.

### **IP with Displacement**

The IP with displacement addressing mode is often used with load and store instructions to make them IP relative.

Note that with this mode the displacement plus a constant of 8 is added to the IP of the instruction.





---

# *Instruction-Set Summary*

**6**

---

Instruction

---

## CHAPTER 6 INSTRUCTION-SET SUMMARY

This chapter provides an overview of the instruction set for the 80960MC processor. Included is a discussion of the instruction format and a summary of the instruction groups and the instructions in each group.

Chapter 17 gives detailed descriptions of each of the instructions. The instructions are listed in this chapter in alphabetical order. Included for each instruction are the assembly-language format, the action taken when the instruction is executed, and examples of how the instruction might be used.

Appendix C provides a detailed description of the factors that affect instruction timing. It also gives the number of clock cycles required for each instruction.

### INSTRUCTION FORMATS

Instructions are described in this reference manual in two formats: assembly language and machine level.

#### Assembly-Language Format

Throughout most of this manual, the instructions are referred to by their assembly-language mnemonics. For example, the add ordinal instruction is referred to as the **addo** instruction.

An assembly-language statement consists of an instruction mnemonic, followed by from 0 to 3 operands, separated by commas. The following example shows the assembly-language statement for the **addo** instruction:

```
addo g5, g9, g7
```

Here, the ordinal operands in global registers g5 and g9 are added together and the result is stored in g7.

A detailed description of the nomenclature used to describe assembly-language instructions is given in Chapter 17.

#### Machine Formats

At the machine level of the processor, all instructions are word aligned. Most of the instructions are one word long, although some memory-addressing modes make use of a two-word format.

There are four instruction formats: register (REG), compare and branch (COBR), control (CTRL), and memory (MEM). Each instruction uses one of these formats, which is determined by the opcode field of the instruction.

The machine-level formats for the instructions are described in detail in Appendix B.

## INSTRUCTION GROUPS

The 80960MC processor implements all the instructions in the 80960 instruction set, which includes all of the data-movement, arithmetic, logical, and program-control instructions commonly found in computer architectures. The processor also includes a set of floating-point instructions and several instructions to handle architectural extensions found in the processor.

The 80960 instruction set is made up of the following groups of instructions:

- Data Movement
- Arithmetic (Ordinal and Integer)
- Logical
- Bit, Bit Field, and Byte
- Comparison
- Branch
- Call/Return
- Fault
- Debug
- Atomic
- Processor Management

The instruction-set extensions found in the 80960MC processor include the following groups of instructions:

- Integer-to-Real Conversion
- Floating Point
- Process Management
- String
- Decimal

Tables 6-1 and 6-2 give a summary of the 80960 instructions and the 80960MC instruction-set extensions, respectively. The actual number of instructions is greater than those shown in this list, because for some operations, several different instructions are provided to handle different operand sizes, data types, or branch conditions.

Table 6-1: Summary of the 80960 Instruction Set

Data Movement	Arithmetic	Logical	Bit, Bit Field, and Byte
Load	Add	And	Set Bit
Store	Subtract	Not And	Clear Bit
Move	Multiply	And Not	Not Bit
Load Address	Divide	Or	Check Bit
	Extended	Exclusive Or	Alter Bit
	Multiply	Not Or	Scan For Bit
	Extended	Or Not	Scan Over Bit
	Divide	Nor	Extract
	Remainder	Exclusive Nor	Modify
	Modulo	Not	Scan Byte For
	Shift	Nand	Equal
	Rotate		
Comparison	Branch	Call/Return	Fault
Compare	Unconditional	Call	Conditional Fault
Conditional	Branch	Call Extended	Synchronize Faults
Compare	Conditional Branch	Call System	
Compare and	Compare and	Return	
Increment	Branch	Branch and Link	
Compare and	Test Condition		
Decrement	Code		
Debug	Atomic	Processor	
Modify Trace	Atomic Add	Flush Local	
Controls	Atomic Modify	Registers	
Mark		Modify Arithmetic	
Force Mark		Controls	
		Modify Process	
		Controls	



Table 6-2: Summary of the 80960MC Instruction-Set Extensions

Conversion	Floating Point	Process Control
Convert Real-to-Integer	Move Real	Schedule Process
Convert Integer-to-Real	Add	Save Process
	Subtract	Resume Process
	Multiply	Load Process Time
	Divide	Signal
	Remainder	Wait
	Scale	Conditional Wait
	Round	Send
	Square Root	Receive
	Sine	Conditional Receive
	Cosine	Send Service
	Tangent	
	Arctangent	
	Log	
	Log Binary	
	Log Natural	
	Exponent	
	Classify	
	Copy Real Extended	
	Compare	
String	Decimal	Miscellaneous
Move String	Move	Inspect Access
Move Quick String	Add With Carry	Load Physical Address
Fill String	Subtract With Carry	Synchronous Move
Compare String		Synchronous Load

The following sections give a brief overview of the instructions in each of these groups. The floating-point instructions are described in Chapter 7.

## DATA MOVEMENT

The data movement instructions include those instructions that move data from memory to the global and local registers; that move data from the global and local registers to memory; and that move data among these registers.

## Load

evoM

The load instructions (listed below) copy bytes or words from memory to a selected register or group of registers:

<b>ld</b>	load	move word	mov
<b>ldob</b>	load byte ordinal	move long word	movl
<b>ldos</b>	load short ordinal	move triple word	movt
<b>ldib</b>	load byte integer	move quad word	movq
<b>ldis</b>	load short integer		
<b>ldl</b>	load long		
<b>ldt</b>	load triple		
<b>ldq</b>	load quad		

For the **ld**, **ldob**, **ldos**, **ldib**, and **ldis** instructions, a memory address and a register are specified in the instruction and the value at the memory address is copied into the register. Zero and sign extending is performed automatically for byte and short (half-word) operands.

The **ld**, **ldl**, **ldt**, and **ldq** instructions copy 4, 8, 12, and 16 bytes from memory into successive registers.

## NOTE

When using the load, store, and move instructions that move 8, 12, or 16 bytes at a time, the rules for register alignment must be followed. Refer to the section in Chapter 3 titled "Register Alignment" for a discussion of these rules.

## Store

For each load instruction there is a corresponding store instruction (listed below), which copies bytes or words from a selected register or group of registers to memory:

<b>st</b>	store
<b>stob</b>	store byte ordinal
<b>stos</b>	store short ordinal
<b>stib</b>	store byte integer
<b>stis</b>	store short integer
<b>stl</b>	store long
<b>stt</b>	store triple
<b>stq</b>	store quad

For the **st**, **stob**, **stos**, **stib**, and **stis** instructions, a register and memory address are specified in the instruction and the value in the register is copied into memory. For the byte and short instructions, the value in the register is automatically reformatted for the shorter memory location. For the **stib** and **stis** instructions, this reformatting can lead to overflow if the register value is too large to be represented in the shorter memory location.

The **st**, **stl**, **stt**, and **stq** instructions copy 4, 8, 12, and 16 bytes from successive registers into memory.

## Move

The move instructions, listed below, copy data from a register or group of registers to another register or group of registers.

<b>mov</b>	move word
<b>movl</b>	move long word
<b>movt</b>	move triple word
<b>movq</b>	move quad word

These move instructions can only be used to move data among the global and local registers. A set of move-real instructions (**movr**, **movrl**, and **movre**) are provided for moving real number values between the global and local registers and the floating-point registers. The move-real instructions are described in Chapter 7.

## Load Address

The **lda** instruction computes an effective address in the address space from an operand presented in one of the addressing modes. A common use of this instruction is to load a constant into a register.

### NOTE

## ARITHMETIC

Table 6-3 lists all the arithmetic operations for which the 80960MC processor provides instructions and the data types that the instructions operate on. An "X" in this table indicates that the 80960 architecture provides an instruction for the specified operation and data type; an "E" indicates that an 80960MC instruction-set extension provides an instruction for the specified operation and data type. An "E\*" indicates that the specified operation can be performed on the specified data type using 80960MC extended-instruction-set instructions, but that a unique instruction for this operation is not provided. For example, a specific instruction is not provided to add two extended-real values. However, this operation can be carried out with either the add real (**addr**) or the add long real (**addrl**) instruction.

With two exceptions, all the processor's arithmetic operations are carried out on operands in registers. The processor does not provide instructions that perform arithmetic operations on operands in memory.

The two instructions that are exceptions are the **atadd** (atomic add) and **atmod** (atomic modify) instructions, which are discussed later in this chapter.

A summary of the arithmetic instructions for real (floating-point) data types is provided in Chapter 7. The following sections describe the arithmetic instructions for ordinal and integer data types.

Table 6-3: Arithmetic Operations

Arithmetic Operations	Integer	Ordinal	Real	Long Real	Extended Real
Add	X	X	E	E	E*
Subtract	X	X	E	E	E*
Multiply	X	X	E	E	E*
Divide	X	X	E	E	E*
Remainder	X	X	E	E	E*
Modulo	X				
Shift Left	X	X			
Shift Right	X	X			
Shift Right Dividing	X				
Scale			E	E	E*
Round			E	E	E*
Square Root			E	E	E*
Sine			E	E	E*
Cosine			E	E	E*
Tangent			E	E	E*
Arctangent			E	E	E*
Exponent			E	E	E*
Log			E	E	E*
Log Binary			E	E	E*
Log Epsilon			E	E	E*
Classify			E	E	E*
Copy Sign					E
Copy Reversed Sign					E

### Add, Subtract, Multiply, and Divide

The following instructions perform add, subtract, multiply, or divide operations on integers and ordinals:

<b>addi</b>	add integer	X	X
<b>addo</b>	add ordinal	X	X
<b>subi</b>	subtract integer	X	X
<b>subo</b>	subtract ordinal	X	X
<b>muli</b>	multiply integer	X	X
<b>mulo</b>	multiply ordinal	X	X
<b>divi</b>	divide integer		X
<b>divo</b>	divide ordinal		X

These instructions perform operations on one-word operands in registers and store the results in a register.

### Extended Arithmetic

The following four instructions are provided to support extended arithmetic operations to be performed (i.e., arithmetic operations on operands greater than one word in length):

<b>addc</b>	add ordinal with carry
<b>subc</b>	subtract ordinal with carry
<b>emul</b>	extended multiply
<b>ediv</b>	extended divide

The **addc** and **subc** instructions add or subtract two words (contained in registers) plus a condition code bit (used as a carry bit). If the result has a carry, the carry bit in the condition code is set. Also, a second condition code bit is set if the operation would have resulted in an integer overflow condition. (The three-bit condition code is contained in the arithmetic controls as described in Chapter 3.)

These instructions treat the operands as ordinals; however, the indication of overflow in the condition code facilitates a software implementation of extended-integer arithmetic.

The **emul** instruction multiplies two ordinals (each contained in a register), producing long ordinal result (stored in two registers). The **ediv** instruction divides a long ordinal by an ordinal, producing an ordinal quotient and an ordinal remainder.

### Remainder and Modulo

The following instructions divide one operand by another and retain the remainder of the operation:

<b>remi</b>	remainder integer
<b>remo</b>	remainder ordinal
<b>modi</b>	modulo integer

The difference between the remainder and modulo instructions lies in the sign of the result. For the **remi** and **remo** instructions, the result has the same sign as the dividend; for the **modi** instruction, the result has the same sign as the divisor.

## Shift and Rotate

The processor provides the following five shift instructions:

<b>shlo</b>	shift left ordinal
<b>shro</b>	shift right ordinal
<b>shli</b>	shift left integer
<b>shri</b>	shift right integer
<b>shrdi</b>	shift right dividing integer

These instructions shift the operand a specified number of bits to the left or to the right. Bits shifted beyond the register boundary are discarded.

The **shlo** instruction shift zeros in from the least-significant bit, and the **shro** instruction shifts zeros in from the most-significant bit. These instructions are equivalent to **mulo** and **divo** by the power of 2, respectively.

The **shli** instruction shifts zeros in from the least-significant bit; if the bits shifted out are not the same as the sign bit, an overflow fault is generated.

The **shri** instruction performs a conventional arithmetic shift-right operation by shifting the sign bit in from the most-significant bit. When this instruction is used to divide a negative integer operand by the power of 2, however, it produces an incorrect quotient. (The discarding of the bits shifted out has the effect of rounding the result toward negative.)

The **shrdi** instruction is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result if the bits shifted out are non-zero and the operand is negative, which produces the correct result for negative operands.

The **shli** and **shrdi** instructions are equivalent to **muli** and **divi** by the power of 2.

The **rotate** instruction rotates the bits of the operand to the left (toward higher significance) by a specified number of bits. Bits shifted beyond the left boundary of the register (bit 31) appear at the right boundary (bit 0).



**LOGICAL**

The following instructions perform bitwise Boolean operations on the specified operands:

<b>and</b>	A and B
<b>notand</b>	(not A) and B
<b>andnot</b>	A and (not B)
<b>xor</b>	not (A = B)
<b>or</b>	A or B
<b>nor</b>	not (A or B)
<b>xnor</b>	A = B
<b>not</b>	not A
<b>notor</b>	(not A) or B
<b>ornot</b>	A or (not B)
<b>nand</b>	not (A and B)

**BIT AND BIT FIELD**

The bit instructions perform operations on a specific bit in an ordinal operand or on a bit field.

**Bit Operations**

The following instructions operate on a specified bit:

<b>setbit</b>	set bit
<b>clrbit</b>	clear bit
<b>notbit</b>	not bit
<b>chkbit</b>	check bit
<b>alterbit</b>	alter bit
<b>scanbit</b>	scan for bit
<b>spanbit</b>	span over bit

The **setbit**, **clrbit**, and **notbit** instructions set, clear, or complement (toggle) a specified bit in an ordinal.

The **chkbit** instruction causes the condition-code bits to be set according to the state of a specified bit in a register. The condition code is set to 010<sub>2</sub> if the bit is set and 000<sub>2</sub> otherwise.

The **alterbit** instruction alters the state of a specified bit in an ordinal according to the condition code. If the condition code is 010<sub>2</sub>, the bit is set; if the condition code is 000<sub>2</sub>, the bit is cleared.

The **scanbit** and **spanbit** instructions find the most significant set bit and clear bit, respectively, in an ordinal.

## int 1

There are two bit field instructions **extract** and **modify**. The **extract** instruction converts a specified bit field, taken from an ordinal value, into an ordinal value. In essence, this instruction shifts a bit field in a register to the right and fills in the bits to the left of the bit field with zeros.

The **modify** instruction copies bits from one register, under control of a mask, into another register. Only the unmasked bits in the destination register are modified.

## BYTE OPERATIONS

The **scanbyte** instruction performs a byte-by-byte comparison of two ordinals to determine if any two corresponding bytes are equal. The condition code is set according to the results of the comparison.

## CONVERSION

Data can be converted from one length to another by means of the load and store instructions. For example, the **ldis** instruction loads a short integer from memory to a register and automatically converts the integer from a half word to a full word.

The 80960MC extended instruction set provides instructions to perform conversions between integer and real data types. These instructions are described in Chapter 7.

## COMPARISON

The processor provides several types of instructions that are used to compare two operands. The following sections describe the compare instructions for ordinal and integer data types. The compare instructions for real data types are discussed in Chapter 7.

### Compare and Conditional Compare

The compare instructions listed below compare two operands, then set the condition-code bits in the arithmetic controls according to the results.

<b>cmpi</b>	compare integer
<b>cmpo</b>	compare ordinal
<b>concmpi</b>	conditional compare integer
<b>concmpo</b>	conditional compare ordinal

The condition-code bits are set to indicate whether one operand is less than, equal to, or greater than the other operand. (Refer to the section in Chapter 3 titled "Functions of the Arithmetic Controls Bits" for a discussion of meanings of the condition-code bits for conditional operations.)

The **cmpi** and **cmpo** instructions simply compare the two operands and set the condition-code bits accordingly.

The **concmpi** and **concmpo** instructions first check the status of bit 2 of the condition code. If it is not set, the operands are compared as with the **cmpi** and **cmpo** instructions. If bit 2 is set, no comparison is performed and the condition-code bits are not changed.

The conditional compare instructions are provided specifically to optimize two-sided range comparisons to check if A is between B and C (i.e.,  $B \leq A \leq C$ ). Here, a compare instruction (**cmpi** or **cmpo**) is used to check one side of the range (e.g.,  $A \geq B$ ) and a conditional compare instruction (**concmpi** or **concmpo**) is used to check the other side (e.g.,  $A \leq C$ ) according to the result of the first comparison.

## Compare and Increment or Decrement

The following instructions compare two operands, set the condition-code bits according to the results, then increment or decrement one of the operands:

<b>cmpinci</b>	compare and increment integer
<b>cmpinco</b>	compare and increment ordinal
<b>cmpdeci</b>	compare and decrement integer
<b>cmpdeco</b>	compare and decrement ordinal

These instructions are intended for use at the end of iterative loops.

## BRANCH

The branch instructions allow the direction of program flow to be changed by explicitly modifying the IP. The processor provides three types of branch instructions:

- unconditional branch
- conditional branch
- compare and branch

The processor also provides a set of instructions for testing the condition code flags of the arithmetic controls. These instructions can be used in conjunction with the compare instructions and the branch instructions as a alternate means of performing conditional branch, and compare and branch operations.

Most of the branch instructions specify the target IP by specifying a signed displacement to be added to the current IP. Other branch instructions specify the memory address of the target IP using one of the processor's addressing modes. This latter group of instructions are called extended-addressing instructions (e.g., branch extended, branch and link extended).

## Unconditional Branch

The following four instructions are used for unconditional branching:

<b>b</b>	Branch
<b>bx</b>	Branch Extended
<b>bal</b>	Branch and Link
<b>balx</b>	Branch and Link Extended

The **b** and **bx** instructions cause program execution to jump to the specified target IP. As described in Chapter 17, these two instructions perform the same function; however, they use different machine-level instruction formats.

The **bal** and **balx** instructions store the address of the next instruction in a specified register, then jump to the specified target IP. (For the **bal** instruction, the RIP is automatically stored in register `g14`; for the **balx** instruction the location of the RIP is specified with an instruction operand.) As described in Chapter 4, the branch and link instructions provide a method of performing procedure calls that does not use the processor's call/return mechanism. Here, the saved instruction address is used as a return IP.

The **bx** and **balx** instructions can be made IP-relative by using the IP with displacement addressing mode.

## Conditional Branch

With the conditional branch (branch if) instructions, the processor checks the condition-code bits in the arithmetic controls. If these bits match the value specified with the instruction, the processor jumps to the target IP. These instructions use the displacement plus IP method of specifying the target IP:

<b>be</b>	branch if equal
<b>bne</b>	branch if not equal
<b>bl</b>	branch if less
<b>ble</b>	branch if less or equal
<b>bg</b>	branch if greater
<b>bge</b>	branch if greater or equal
<b>bo</b>	branch if ordered
<b>bno</b>	branch if unordered

(Refer to the section in Chapter 3 titled "Functions of the Arithmetic Controls Bits" for a discussion of meanings of the condition-code bits for conditional operations.)

The **bo** and **bno** instructions refer to comparisons of real numbers. Ordered and unordered real numbers are described in Chapter 7.

## Compare and Branch

The compare and branch instructions compare two operands, then branch according to the results. There are three subtypes of instructions in this group: compare integer, compare ordinal, and check bit:

<b>cmpibe</b>	compare integer and branch if equal
<b>cmpibne</b>	compare integer and branch if not equal
<b>cmpibl</b>	compare integer and branch if less
<b>cmpible</b>	compare integer and branch if less or equal
<b>cmpibg</b>	compare integer and branch if greater
<b>cmpibge</b>	compare integer and branch if greater or equal
<b>cmpibo</b>	compare integer and branch if ordered
<b>cmpibno</b>	compare integer and branch if unordered
<b>cmpobe</b>	compare ordinal and branch if equal
<b>cmpobne</b>	compare ordinal and branch if not equal
<b>cmpobl</b>	compare ordinal and branch if less
<b>cmpoble</b>	compare ordinal and branch if less or equal
<b>cmpobg</b>	compare ordinal and branch if greater
<b>cmpobge</b>	compare ordinal and branch if greater or equal
<b>bbs</b>	check bit and branch if set
<b>bbc</b>	check bit and branch if clear

With the compare-ordinal-and-branch and compare-integer-and-branch instructions, two operands are compared and the condition-code bits are set, as with the compare instructions described earlier in this chapter. A conditional branch is then executed as with the conditional branch (branch if) instructions.

With the check-bit-and-branch instructions, one operand specifies a bit to be checked in the other operand. The condition-code bits are set according to the state of the specified bit (i.e.,  $010_2$  if the bit is set and  $000_2$  if the bit is clear). A conditional branch is then executed according to the setting of the condition-code bits.

## Test Condition Codes

The following test instructions allow the state of the condition-code bits to be tested:

<b>teste</b>	test if equal
<b>testne</b>	test if not equal
<b>testl</b>	test if less
<b>testle</b>	test if less or equal
<b>testg</b>	test if greater
<b>testge</b>	test if greater or equal
<b>testo</b>	test if ordered
<b>testno</b>	test if unordered

These instructions cause a TRUE ( $1_2$ ) to be stored in a destination register if the condition code matches the condition specified with the instruction. Otherwise, a FALSE ( $0_2$ ) is stored in the register.

## CALL AND RETURN

The processor offers an on-chip call/return mechanism for making procedure calls to local procedures and kernel procedures. This call/return mechanism is described in detail in Chapter 4. The following four instructions are provided to support this mechanism.

<b>call</b>	call
<b>callx</b>	call extended
<b>calls</b>	call system
<b>ret</b>	return

The **call** and **callx** instructions call local procedures. The **call** instruction specifies the target procedure (the first instruction of the procedure) by adding a signed displacement to the IP. The **callx** instruction uses extended addressing, as described for the **bx** and **balx** instructions, to specify the target procedure. For both of these instructions, a new set of local registers and a new stack frame are allocated for the called procedure.

The **calls** instruction operates similarly to the **call** and **callx** instructions, except that it gets its target procedure address from the system procedure table. An index number included as an operand in the instruction provides an entry point into the procedure table.

Depending on the type of entry being pointed to in the procedure table, the **calls** instruction can cause a supervisor call to be executed. A supervisor call causes the processor to switch to the supervisor stack and to switch to supervisor mode. The supervisor call is described in detail in Chapter 4.

The **ret** instruction performs a return from a called procedure to the calling procedure (the procedure that made the call). This instruction obtains its target IP (return IP) from linkage information that was saved for the calling procedure. The **ret** instruction is used to return from local and supervisor calls and from implicit calls to interrupt and fault handlers.

## CONDITIONAL FAULTS

Generally, the processor generates faults automatically as the result of certain operations. Fault handling routines are then invoked to handle the various types of faults without explicit intervention by the currently running process. (Faults are discussed in detail in Chapter 12.)

The following conditional fault instructions permit a fault to be generated explicitly according to the state of the condition-code bits:

<b>faulte</b>	fault if equal
<b>faultne</b>	fault if not equal
<b>faultl</b>	fault if less
<b>faultle</b>	fault if less or equal
<b>faultg</b>	fault if greater
<b>faultge</b>	fault if greater or equal
<b>faulto</b>	fault if ordered
<b>faultno</b>	fault if unordered



The synchronize faults (**syncf**) instruction is provided to force all faults to be precise in situations when the processor is executing two instructions in parallel. The function and use of this instruction is discussed in detail in the section in Chapter 12 titled "Precise and Imprecise Faults."

## DEBUG

The processor supports debugging and monitoring of program activity through the use of trace events. The following instructions support these debugging and monitoring tools:

<b>modtc</b>	modify trace controls
<b>mark</b>	mark
<b>fmark</b>	force mark

The trace functions are controlled through the processor's trace controls bits. Some of these bits allow various types of tracing to be enabled or disabled. Other bits act as flags to indicate when an enabled trace event has been detected. (Trace controls are described in detail in Chapter 16.)

The **modtc** instruction permits the trace controls bits to be modified.

The **mark** instruction causes a breakpoint trace event to be generated if the breakpoint trace mode is enabled. The **fmark** instruction generates a breakpoint trace independent of the state of the breakpoint trace mode flag. The latter two instructions allow a breakpoint to be placed anywhere in a program.

## ATOMIC INSTRUCTIONS

The atomic instructions perform read-modify-write operations on operands in memory. They insure that when one atomic operation is performed on a specific block of memory it will be completed before another atomic operation can be performed on the same block. These instructions are particularly useful in systems that use multiple processors where all of the processors have access to system memory.

There are two atomic instructions: atomic add (**atadd**) and atomic modify (**atmod**). The **atadd** instruction causes an operand to be added to the value in the specified memory location. The **atmod** causes bits in the specified memory location to be modified under control of a mask.

## PROCESSOR MANAGEMENT

The processor provides several instructions for use in controlling processor-related functions.

The **modpc** instruction provides a method of reading and modifying the contents of the process controls.

In certain instances, it is necessary to insure that the contents of the local-register save area of the stack frames are the same as the local registers. The flush local registers instruction

(**flushreg**) automatically stores the contents of all the local register sets, except the current set, in the register save area of their associated stack frames.

The arithmetic controls cannot be addressed with the load, move, and store instructions or the bit instructions. Instead, special instructions are provided for this purpose.

The modify arithmetic controls instruction (**modac**) permits bits in the arithmetic controls register to be modified under the control of a mask.

## 80960MC NON-FLOATING-POINT INSTRUCTION-SET EXTENSIONS

The following non-floating-point instructions are extensions to the 80960-architecture instruction set. These instructions are provided to support extended non-floating-point features such as string operations, decimal arithmetic, multiprocessing, process management, and virtual memory management.

### Process Management

The processor provides several instructions for use in process management. These instructions do not dictate a particular process management scheme. Instead they provide support for a wide variety of process management mechanisms. These instructions can be divided into two groups: process control and interprocess communication.

The processor must be in the supervisor mode to execute the process management instructions. Process management is described in detail in Chapters 13 and 14.

### Process Control

The following instructions provide process control services:

<b>saveprcs</b>	save process
<b>resumpcrs</b>	resume process
<b>schedprcs</b>	schedule process
<b>ldtime</b>	load process time

The processor defines two data structures for use in process control: a process control block (PCB) and a dispatching port. The PCB is used to maintain information about the process such as the status of the execution environment when the process was last suspended and system resources allocated to the process. The dispatching port is used for queuing processes that are waiting to be worked on by the processor.

The **resumpcrs** instruction causes the processor to switch to the specified process. The **saveprcs** instruction causes the current state of the currently running process to be saved in the PCB.

These two instructions perform roughly the same functions as the RESUME and SAVE functions of most UNIX™ kernels. A dispatching port is not needed with these instructions.

The **schedprcs** instruction causes a process to be enqueued at a dispatching port.

The processor provides a mechanism for keeping track of process execution time. The **ldtime** instruction supports this mechanism by providing a method of loading the elapsed execution time of the currently running process into a specified register.

The **modpc** instruction, which is described earlier in this chapter, provides a method of reading and modifying the contents of the process controls for the currently running process.

## Interprocess Communication

The processor supports two techniques for communication among processes: semaphores and communication ports.

**Semaphores.** A semaphore is essentially a queue for synchronizing the activities of interdependent processes. The following instructions support communication through semaphores:

<b>wait</b>	wait
<b>condwait</b>	conditional wait
<b>signal</b>	signal

The **wait** instruction causes the processor to check the specified semaphore for a signal, in the form of a semaphore count. If the semaphore count is non-zero, the processor decrements the count and continues execution of the current process. If the count is zero, the processor suspends the current process and queues it to the semaphore. The process is then said to be blocked.

The **condwait** instruction performs the same function as the **wait** instruction, except that if a signal is not found, the process is not blocked. Instead, the condition-code bits are set to indicate whether or not the signal was received.

The **signal** instruction causes the processor to send a signal to the specified semaphore. If processes are queued at the semaphore, the first process in the queue is unblocked. Otherwise, the semaphore count is incremented by one.

**Communication Ports.** A communication port is similar to a semaphore except that it also provides a message-passing mechanism. A communication port can thus be used both for synchronizing processes and as a means of passing messages among processes.

Messages are one word long. This message word can contain anything. Commonly, it contains a one word message, a process number, or the address of a longer message.

The following instructions support communication ports:

<b>receive</b>	receive
<b>condrec</b>	conditional receive
<b>send</b>	send
<b>sendserv</b>	send service

With the **receive** instruction, the processor checks the specified communication port for a message. If a message is queued at the port, it loads the message into a specified register and continues execution of the current process. If the message queue is empty, the processor suspends the current process and queues it at the communication port (i.e., blocks the process).

The **condrec** instruction is similar to the **receive** instruction except that the process is not blocked if the message queue is empty. Instead the processor sets the condition-code bits to indicate whether or not the receive operation was carried out.

The **send** instruction causes the processor to send a message to a specified communication port. If there are no processes at the port for messages, the processor enqueues the message at the port and continues executing the current process. If there are queued processes at the port, the first process in the queue is unblocked, given the message, and rescheduled at the dispatching port. The processor then resumes execution of the current process.

The **sendserv** instruction causes the processor to suspend the current process and send it as a message to the specified communication port.

## String

The 80960MC extended instruction set provides the following string instructions perform operations on byte strings in memory:

<b>movstr</b>	move string
<b>movqstr</b>	move quick string
<b>fill</b>	fill string
<b>cmpstr</b>	compare string

The **movstr** and **movqstr** instructions move a byte string from one location in memory to another. These instructions operate identically except that the **movstr** instruction guarantees that if the strings overlap, no byte in the source string is overwritten until it is copied to the destination string. If the strings being moved do not overlap, the **movqstr** instruction should be used because it performs the move operation faster.

The **fill** instruction copies an ordinal operand repeatedly into a byte string in memory.

The **cmpstr** instruction compares two byte strings of equal length, then sets the condition-code bits to show whether or not the strings are identical.

## Decimal

The following three instructions are provided for use in decimal-arithmetic algorithms:

<b>dmovt</b>	move and test decimal
<b>daddc</b>	decimal add with carry
<b>dsubc</b>	decimal subtract with carry

These instructions operate on 32-bit decimal operands that contain an 8-bit, ASCII-coded decimal in the least-significant byte of the word (as shown in Figure 5-3).

The **dmovt** instruction moves a decimal operand from one register to another and tests the least significant byte of the operand to determine if it is a decimal digit (0 to 9). It sets the condition code according to the results of the test: 010<sub>2</sub> if the operand contains a decimal digit and 000<sub>2</sub> otherwise.

The **daddc** and **dsubb** instructions operate similarly to the **addc** and **subc** instructions. They add or subtract two decimal digits plus bit 1 of the condition code (used as a carry-in bit). If the operation produces a decimal carry, the condition code is set accordingly. The subtraction operation is carried out in 10's complement arithmetic.

These instructions can be used iteratively to add or subtract decimal values of any length.

With the 80960MC processor, the most efficient method of multiplying or dividing decimal numbers is to convert them into extended-real numbers and use the **mulr** and **divr** instructions. Decimal values of up to 18 decimal digits can be handled with this technique.

### Miscellaneous Instructions

The following instructions perform two special synchronous operations on operands in memory and two memory management functions.

### Synchronous Load and Move

The processor's store instructions are executed asynchronously with the memory controller. Once the processor sends data out on its bus for storage in main memory, it continues with the next instruction in the instruction stream, assuming that its bus control logic will carry out the operation.

The 80960MC processor provides four special instructions for performing memory operations that perform store and move operations synchronously with memory.

The synchronous load instruction (**synld**) loads a word from memory into a register. When this instruction is performed, the processor waits until a condition code bit is set in the arithmetic controls, indicating that the operation has been completed, before it begins executing the next instruction. The **synld** instruction is used primarily to read the contents of the interrupt-control register, as described in Chapter 10.

The synchronous move instructions (**synmov**, **synmovl**, and **synmovq**) perform synchronous moves of data from one location in memory to another. These instructions are used primarily for sending IAC messages, as described in Chapter 11.

### Memory-Management Functions

The inspect access instruction (**inspac**) returns the respective page rights of a specified page. This instruction is used in memory management routines.

The load physical address (**ldphy**) instruction translates an address in the address space into a physical address. The primary function of this instruction is to translate virtual addresses into physical addresses.







Figure 7-10

---

## CHAPTER 7

# FLOATING-POINT OPERATION

This chapter describes the floating-point processing capabilities of the 80960MC processor. The subjects discussed include the real number data types, the execution environment for floating-point operations, the floating-point instructions, and fault and exception handling.

### INTRODUCING THE 80960MC FLOATING-POINT ARCHITECTURE

The floating-point architecture used in the 80960MC processor is designed to allow a convenient implementation of the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. This hardware architecture, along with a small amount of software support, conforms to the IEEE standard and provides support for the following data structures and operations:

- Real (32-bit), long-real (64-bit), and extended-real (80-bit) floating-point number formats.
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversion between integer and floating-point formats
- Conversion between different floating-point formats
- Handling of floating-point exceptions, including non-numbers (NaNs)

The software to support the 80960MC floating-point architecture is needed primarily to handle conversions between real numbers and decimal strings.

In addition, the 80960MC floating-point architecture supports several functions that go beyond the IEEE standard. These functions fall into two categories:

- functions recommended in the appendix to the IEEE standard, such as copy sign and classify, and
- commonly used transcendental functions, including trigonometric, logarithmic, and exponential functions.

### REAL NUMBERS AND FLOATING-POINT FORMAT

This section provides an introduction to real numbers and how they are represented in floating-point format. Readers who are already familiar with numeric processing techniques and the IEEE standard may wish to skip this section.

#### Real Number System

As shown at the top of Figure 7-1, the real-number system comprises the continuum of real numbers from minus infinity ( $-\infty$ ) to plus infinity ( $+\infty$ ).

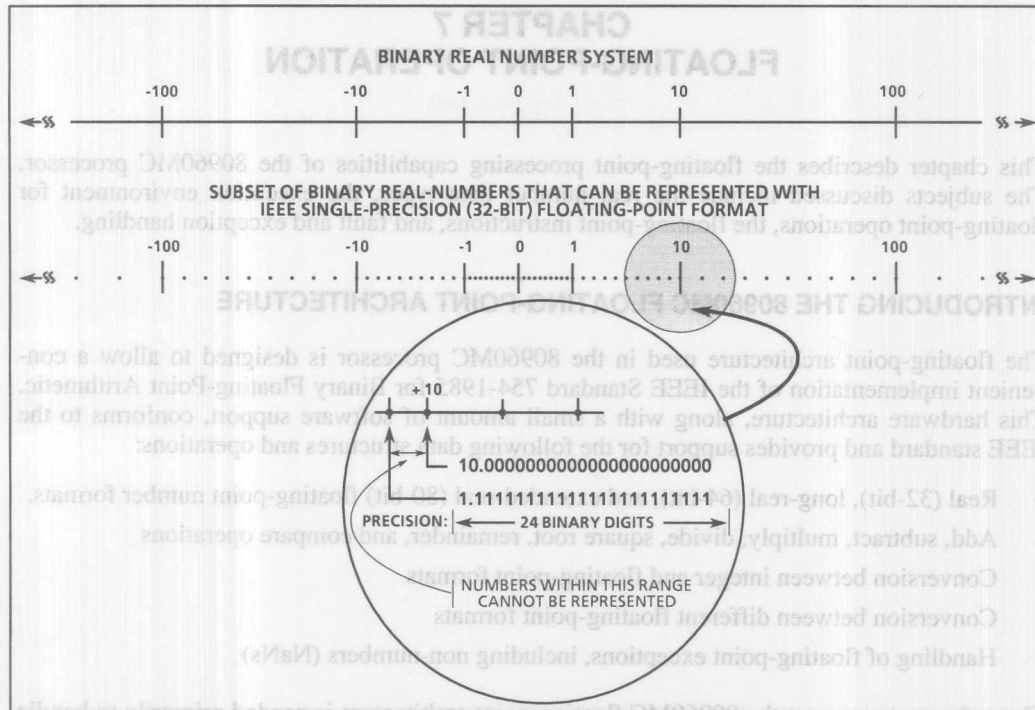


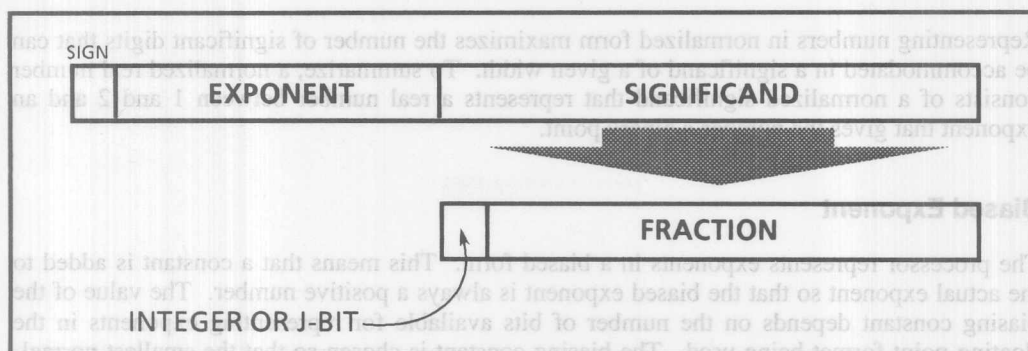
Figure 7-1: Binary Number System

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number calculations. As shown at the bottom of Figure 7-1, the subset of real numbers that a particular processor supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the format that the processor uses to represent real numbers.

## Floating-Point Format

To increase the speed and efficiency of real number computations, computers or numeric processors typically represent real numbers in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent. Figure 7-2 shows the binary floating-point format that the processor uses. This format conforms to the IEEE standard.

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a one-bit binary integer (also referred to as the j-bit) and a binary fraction. The j-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power that the significand is raised to.



**Figure 7-2: Binary Floating-Point Format**

Table 7-1 shows how the real number 201.187 (in ordinary decimal format) is stored in floating-point format. The table lists a progression of real number notations that leads to the format that the 80960MC processor uses. In this format, the binary real number is normalized and the exponent is biased.

**Table 7-1: Real-Number Notation**

NOTATION	VALUE		
ORDINARY DECIMAL	201.187		
SCIENTIFIC DECIMAL	$2.01187\text{E}_{102}$		
SCIENTIFIC BINARY	$1.1001001001011111\text{E}_{2111}$		
SCIENTIFIC BINARY (BIASED EXPONENT)	$1.1001001001011111\text{E}_{210000110}$		
32-BIT FLOATING-POINT FORMAT (NORMALIZED)	SIGN	BIASED EXPONENT	SIGNIFICAND
	0	10000110	1001001001011111 1. (IMPLIED)

### Normalized Numbers

In most cases, the processor represents real numbers in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and a fraction as follows:

$$1.\text{fff}...\text{ff}$$

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)



Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that gives the number's binary point.

### Biased Exponent

The processor represents exponents in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

(The biasing constants for the various sizes of real data types that the processor supports are given in the section later in this chapter titled "Real Data Types".)

### Real Number and Non-Number Encodings

The real numbers that are encoded in the floating-point format described above are generally divided into three classes:  $\pm 0$ ,  $\pm$  nonzero-finite numbers, and  $\pm \infty$ . Encodings for non-numbers (NaNs) are also defined. The term NaN stands for "Not a Number."

Figure 7-3 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single-precision (32-bit) format, where the term "s" indicates the sign bit, "e" the biased exponent, and "f" the fraction. (The exponent values are given in decimal.)

### Signed Zeros

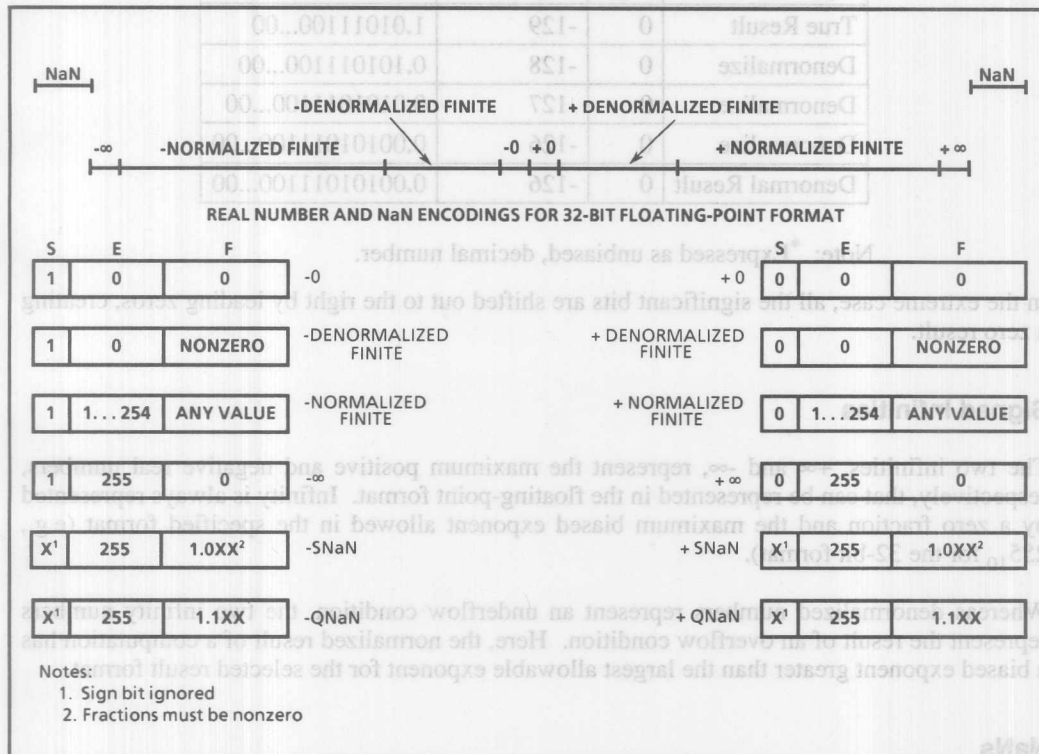
Zero can be represented as a +0 or a -0 depending on the sign-bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an  $\infty$  that has been reciprocated.

### Signed, Nonzero, Finite Values

The class of signed, nonzero, finite values is divided into two groups: normalized and denormalized. The normalized finite numbers comprise all the nonzero finite values that can be encoded in a normalized real number format from zero to  $\infty$ . In the 32-bit form shown in Figure 7-3, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254<sub>10</sub> (unbiased, the exponent range is from -126<sub>10</sub> to +127<sub>10</sub>).

## Denormalized Numbers

When real numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.



**Figure 7-3: Real Numbers and NaNs**

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called denormalized numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization causes a loss of precision (the number of significant bits in the fraction is reduced by the leading zeros).

When performing normalized floating-point computations, a processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an underflow condition.

A denormalized number is computed through a technique called gradual underflow. Table 7-2 gives an example of gradual underflow in the denormalization process. Here the 32-bit format is being used, so the minimum exponent (unbiased) is  $-126_{10}$ . The true result in this example requires an exponent of  $-129_{10}$  in order to have a normalized number. Since  $-129_{10}$  is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of  $-126_{10}$  is reached.

Table 7-2: Denormalization Process

Operation	Sign	Exponent*	Significand
True Result	0	-129	1.01011100...00
Denormalize	0	-128	0.101011100...00
Denormalize	0	-127	0.0101011100...00
Denormalize	0	-126	0.00101011100...00
Denormal Result	0	-126	0.00101011100...00

Note: \*Expressed as unbiased, decimal number.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

### Signed Infinities

The two infinities,  $+\infty$  and  $-\infty$ , represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a zero fraction and the maximum biased exponent allowed in the specified format (e.g.,  $255_{10}$  for the 32-bit format).

Whereas denormalized numbers represent an underflow condition, the two infinity numbers represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

### NaNs

Since NaNs are non-numbers, they are not part of the real number line. In Figure 7-3, the encoding space for NaNs in the 80960MC floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction. (The sign bit is ignored for NaNs.)

The IEEE standard defines two specific NaN values: a quiet NaN (QNaN) and a signaling NaN (SNaN). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs signal an invalid-operation exception whenever they appear as operands in arithmetic operations. Exceptions are discussed later in this chapter in the section titled "Exceptions and Fault Handling."

The section at the end of this chapter titled "Operations on NaNs" provides detailed information on how the processor handles NaNs.

## REAL DATA TYPES

The processor supports three real-number data formats: real, long real, and extended real. These formats correspond directly to the single-precision, double-precision, and double-extended precision formats in the IEEE standard. Figure 7-4 shows these data formats and gives the resolution that each provides.

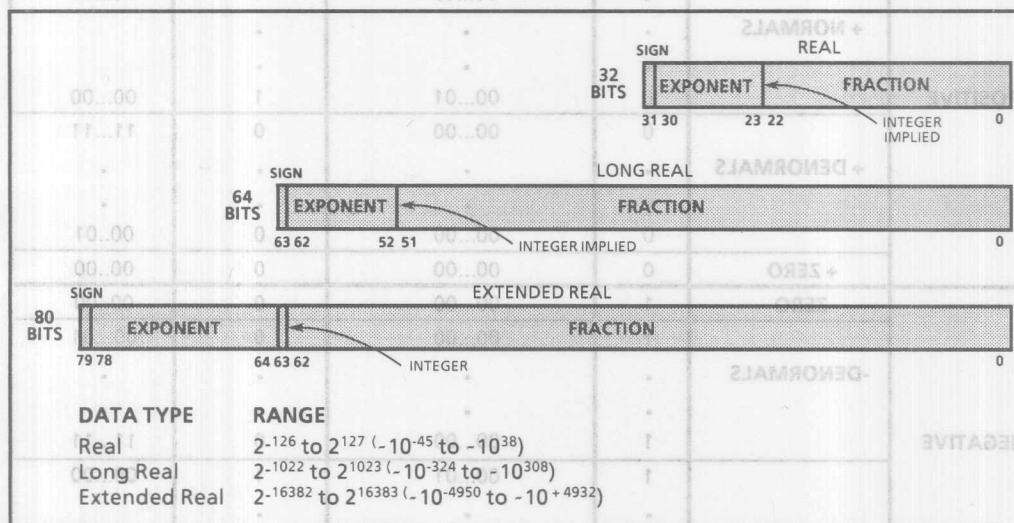


Figure 7-4: Real-Number Formats

As described earlier in this chapter, the processor represents exponents in a biased format. For real values, the biasing constant is 127; for long-real values, it is 1023; and for extended-real values, it is 16383.

For the real and long-real formats, only the fraction is given for the significand. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers. For the extended-real format, the integer is contained in bit 63, and the most-significant fraction bit is bit 62. Here, the integer is explicitly set to 1 for normalized numbers, infinities, and NaNs, and to 0 for zero and denormalized numbers. A non-zero exponent with the integer bit set to zero is a reserved encoding, which will result in a floating reserved-encoding exception being signaled.

Table 7-3 shows the encodings for all the classes of real numbers (i.e., zero, denormalized finite, normalized finite, and  $\infty$ ) and NaNs, for each of the three real data-types.

## EXECUTION ENVIRONMENT FOR FLOATING-POINT OPERATIONS

An important feature of the 80960MC processor is that the floating-point processing capabilities have been integrated into the execution environment of the processor. Operations on floating-point numbers are carried out using the same registers that are used for ordinals and integers. In addition, four floating-point registers have been provided for extended-precision floating-point arithmetic. The following sections describe how floating-point operations are handled in the processor's execution environment.

Table 7-3: Real Numbers and NaN Encodings

	CLASS	SIGN	BIASED EXPONENT	INTEGER <sup>1</sup>	FRACTION
POSITIVE	$+\infty$	0	11...11	1	00...00
		0	11...10	1	11...11
	+ NORMALS	.	.	.	.
		.	.	.	.
		0	00...01	1	00...00
	+ DENORMALS	0	00...00	0	11...11
NEGATIVE		.	.	.	.
		.	.	.	.
	- ZERO	0	00...00	0	00...01
		0	00...00	0	00...00
	- DENORMALS	1	00...00	0	00...01
		1	00...00	0	11...11
		1	00...01	1	00...00
	- NORMALS	.	.	.	.
NaN	SNaN	X	11...11	1	0X...XX <sup>2</sup>
	QNaN	X	11...11	1	1X...XX
			REAL:	8 BITS	23 BITS
			LONG REAL:	11 BITS	52 BITS
			EXTENDED REAL:	15 BITS	63 BITS

## Notes:

1. Integer is implied for real and long real formats and is not stored.
2. Fraction for SNaN must be non-zero.

## Registers

All of the registers in the processor's execution environment, (i.e., global, local, and floating point) can be used for floating-point operations. When using global or local registers, real values (i.e., 32 bits) are contained in one register; long-real values (i.e., 64 bits) are contained in two successive registers; and extended-real values (i.e., 80 bits) are contained in three successive registers.



Figure 7-5 shows how the three forms of the real data type are encoded when stored in global and local registers. Note that long-real values must be aligned on even-numbered register boundaries (e.g., g0, g2, ...). Extended-real values must be aligned on register boundaries that are an integral multiple of four (e.g., g0, g4, ...).

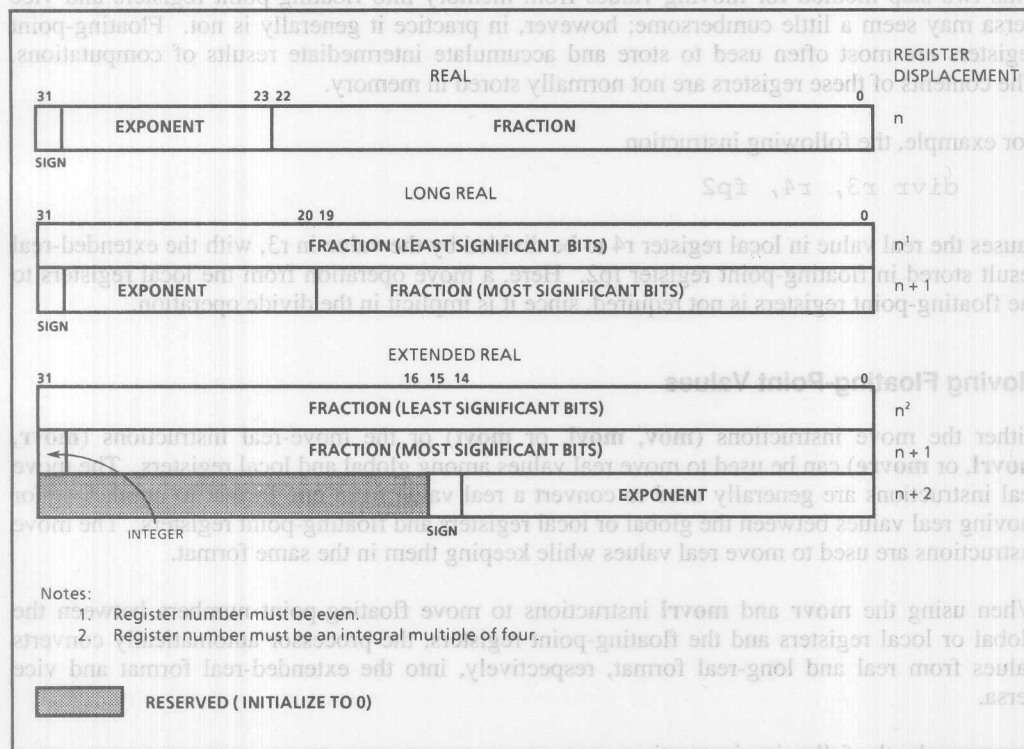


Figure 7-5: Storage of Real Values in Global and Local Registers

Real values in the floating-point registers are always in the extended-real format. When a real or long-real value is moved from global or local registers to a floating-point register, the processor automatically reformats it for the extended-real format.

## Loading and Storing Floating-Point Values

Floating-point values are loaded from memory into global or local registers using the load (**ld**), load long (**ldl**), and load triple (**ldt**) instructions. Likewise, floating-point values in global or local registers are stored in memory using the store (**st**), store long (**stl**), and store triple (**stt**) instructions.

Loading a floating-point value into a floating-point register requires two steps (two instructions). First, a floating-point value must be loaded from memory into one or more global or local registers. Then, the value must be moved to the floating-point register using a move real (**movr**), move long-real (**movrl**), or move extended-real (**movre**) instruction.



A similar two-step procedure is required to store a value from a floating-point register into memory. The value must first be moved into one or more global or local registers (using a **movr**, **movrl**, or **movre** instruction), then stored in memory.

This two-step method for moving values from memory into floating-point registers and vice versa may seem a little cumbersome; however, in practice it generally is not. Floating-point registers are most often used to store and accumulate intermediate results of computations. The contents of these registers are not normally stored in memory.

For example, the following instruction

```
divr r3, r4, fp2
```

causes the real value in local register r4 to be divided by the value in r3, with the extended-real result stored in floating-point register fp2. Here, a move operation from the local registers to the floating-point registers is not required, since it is implicit in the divide operation.

### Moving Floating-Point Values

Either the move instructions (**mov**, **movl**, or **movt**) or the move-real instructions (**movr**, **movrl**, or **movre**) can be used to move real values among global and local registers. The move real instructions are generally used to convert a real value from one format to another or for moving real values between the global or local registers and floating-point registers. The move instructions are used to move real values while keeping them in the same format.

When using the **movr** and **movrl** instructions to move floating-point numbers between the global or local registers and the floating-point registers, the processor automatically converts values from real and long-real format, respectively, into the extended-real format and vice versa.

For example, the following instruction

```
movr g3, fp1
```

causes a 32-bit, real value in global register g3 to be converted to 80-bit, extended-real format and placed in floating-point register fp1.

Going the opposite direction, the instruction

```
movrl fp0, r4
```

causes an extended-real value in floating-point register fp0 to be converted to 64-bit, long-real format and placed in local registers r4 and r5.

The **movre** instruction moves 80-bit, extended-real values between registers, without format conversion. When this instruction is used to move a value from three global or local registers to a floating-point register, the processor extracts the 80-bit value from the three word extended-real format. When moving a value from a floating-point register to global or local registers, the processor inserts the 80-bit value into the three registers in the three-word format.

## Arithmetic Controls

The arithmetic controls are used extensively to control the arithmetic and faulting properties of floating-point operations. Table 7-4 shows the bits in the arithmetic controls that are used in floating-point operations.

**Table 7-4: Arithmetic Controls Used in Floating-Point Operations**

Arithmetic Control Bits	Function
0 - 2	Condition code
3 - 6	Arithmetic status field
8	Integer overflow flag
12	Integer overflow mask
16	Floating overflow flag
17	Floating underflow flag
18	Floating invalid-operation flag
19	Floating zero-divide flag
20	Floating inexact flag
24	Floating overflow mask
25	Floating underflow mask
26	Floating invalid-operation mask
27	Floating zero-divide mask
28	Floating inexact mask
29	Normalizing mode flag
30 - 31	Rounding control

The condition code flags are used to indicate the results of comparisons of real numbers, just as they are for integers and ordinals.

The arithmetic status field is used to record results from the classify real (**classr** and **classrl**) and remainder real (**remr** and **remrl**) instructions. These instructions are discussed later in this chapter.

The floating-point flags indicate exceptions to floating-point operations. Here, the term exception refers to a potentially undesirable operation (such as dividing a number by zero) or an undesirable result (such as underflow). The flags provide a means of recording the occurrence of specific exceptions.

The floating-point masks provide a method of inhibiting the processor from invoking a fault handler when an exception is detected.

Use of the floating-point flag and mask bits are discussed later in this chapter in the section titled "Exceptions and Fault Handling."

### Normalizing Mode

The normalizing-mode flag specifies whether the processor operates in normalizing mode (set) or not (clear).

Normalizing mode is the most common mode of operation. Here, the processor operates on valid floating-point operands, regardless of whether they are normalized or denormalized values.

When the processor is not operating in normalizing mode, it signals a reserved-encoding exception whenever it encounters a denormalized floating-point value as a source operand. In either mode, denormalized numbers are produced if the underflow exception is masked.

There are no flag or mask bits in the arithmetic controls for this exception. When a reserved-encoding exception is detected, the processor generates a floating reserved-encoding fault and leaves the destination operand unchanged (i.e., no result is stored).

The unnormalized mode of operation is provided to allow unnormalized arithmetic to be simulated with software. Here, a fault handler routine can be used to perform unnormalized arithmetic whenever a reserved-encoding exception is signaled.

### Rounding Control

Often the infinitely precise result of an arithmetic operation cannot be encoded exactly in the format of the destination operand. For example, the following value has a 24-bit fraction. The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the real (32-bit) format:

1.0001 0000 1000 0011 1001 0111E<sub>2</sub> 101

The processor must then round the result to one of the following two values:

1.0001 0000 1000 0011 1001 011E<sub>2</sub> 101

1.0001 0000 1000 0011 1001 100E<sub>2</sub> 101

A rounded result is called an inexact result. When an inexact result is produced, the floating-point inexact flag bit in the arithmetic controls is set.

The processor rounds results according to the destination format (real, long real, or extended real) and the setting of the rounding-mode flags of the arithmetic controls. Four types of rounding are allowed, as described in Table 7-5.

**Table 7-5: Rounding Methods**

Rounding Mode	Description
Round up (toward $+\infty$ )	Rounded result is close to but no less than the infinitely precise result
Round down (toward $-\infty$ )	Rounded result is close to but no greater than the infinitely precise result
Round toward zero (Truncate)	Rounded result is close to but no greater in absolute value than the infinitely precise result
Round to nearest (even)	Rounded result is close to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the one with the least-significant bit of zero).

When the infinitely precise result is between the largest positive finite value allowed in a particular format and  $+\infty$ , the processor rounds the result as shown in Table 7-6.

**Table 7-6: Rounding of Positive Numbers**

Rounding Mode	Description
Round up (toward $+\infty$ )	$+\infty$
Round down (toward $-\infty$ )	Maximum, positive finite value
Round toward zero (Truncate)	Maximum, positive finite value
Round to nearest (even)	$+\infty$

When the infinitely precise result is between the largest negative finite value allowed in a particular format and  $-\infty$ , the processor rounds the result as shown in Table 7-7.

**Table 7-7: Rounding of Negative Numbers**

Rounding Mode	Description
Round up (toward $+\infty$ )	Maximum, negative finite value
Round down (toward $-\infty$ )	$-\infty$
Round toward zero (Truncate)	Maximum, negative finite value
Round to nearest (even)	$-\infty$

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

The floating-point instructions allow a result to be stored in a shorter destination than the source operands. For example, the instruction

```
addr fp1, fp2, g5
```

produces a real (32-bit) result from two extended-real (80-bit) source operands. In all such operations, only one rounding error occurs: the error that occurs when rounding the infinitely precise result to the size of the destination format.

Technically, an operation which computes a narrow result from wide operands is in violation of the IEEE standard. However, systems that are designed to conform to the IEEE standard do not need to use this capability of the processor.

## INSTRUCTION FORMAT

The instruction format for floating-point instructions is the same as for the other processor instructions. When programming in assembly language, an assembly language statement begins with an instruction mnemonic and is followed by from one to three operands. For example, the multiply-real instruction **mulr** might be used as follows:

```
mulr r8, r9, fp3
```

Here, real operands in local registers r8 and r9 are multiplied together and the result is stored in floating-point register fp3.

From the machine level point of view, all floating-point instructions use the REG format. Refer to Appendix B for details on the REG format instructions.

## INSTRUCTION OPERANDS

Operands for floating-point instructions can be either floating-point literals or registers. The processor recognizes two encodings for floating-point literals: +0.0 and +1.0.

All of the registers in the processor's execution environment (global registers g0 through g15, local registers r0 through r15, and floating-point registers fp0 through fp3) can be used as operands in floating-point instructions. (Of course, registers g15, r0, r1, and r2 would generally not be used for storing floating-point numbers, since they are reserved for stack management functions.)

When global or local registers are specified as operands, the instruction mnemonic (or opcode) determines how the values in these registers are interpreted. For example, there are two floating-point divide instructions: divide real (**divr**) and divide long real (**divrl**). When using the **divr** instruction, the processor assumes that global- or local-register operands contain real (32-bit) values. When using the **divrl** instruction, global- or local-register operands are assumed to contain long-real (64-bit) values.

With either instruction, floating-point registers (containing extended-real values) can also be used as operands.



Using floating-point registers as operands allows mixed format or mixed precision arithmetic to be performed with either real and extended-real values or long-real and extended-real values. Mixed-format operations with real and long-real values are not supported.

## SUMMARY OF FLOATING-POINT INSTRUCTIONS

The processor's floating-point instructions consist of all instructions for which at least one operand is a real data type.

These instructions can be divided into the following groups:

- Data Movement
- Data-Type Conversion
- Basic Arithmetic
- Comparison and Classification
- Trigonometric
- Logarithmic and Exponential

The following sections give a brief overview of the instructions in each group. Detailed descriptions of the operations of these instructions are given in Chapter 17.

### Data Movement

As has been described earlier in this chapter, the non-floating-point load and store instructions are used to move real values between registers and memory. Once in registers, the non-floating-point move instructions (**mov**, **movl**, and **movt**) are used to move real values between global and local registers without format conversion; whereas, the floating-point move instructions (**movr**, **movrl**, and **movre**) are used to move real values between global and local registers and floating-point registers.

The copy-sign-real-extended (**cpysre**) and copy-reverse-sign-real-extended (**cpysrre**) instructions provide a means of copying the sign of one extended-real value to another, if one of the values is in a floating-point register. This operation is best performed on real and long-real values using the bit instructions **chkbit** and **alterbit**.

### Data-Type Conversion

Two types of data-type conversions are provided: conversion from one floating-point format to another (e.g., real to extended real) and conversion between integer and real.

Conversion between floating-point formats is handled in either of two ways: explicitly by move instructions or implicitly by using the floating-point registers as operands in instructions.

As described earlier in this chapter, the **movr** instruction implicitly converts values from real to extended real, and vice versa, when moving values between global or local registers and floating-point registers. Likewise, the **movrl** instruction implicitly converts values from long real to extended real, and vice versa.



Conversion between real and long-real formats requires the use of both instructions. For example, the following two instructions convert a real value in global register g6 to a long-real value contained in g6 and g7, using a floating-point register for intermediate storage of the value:

```
movr g6, fp1
movrl fp1, g6
```

Implicit format conversion is also provided through the arithmetic, trigonometric, logarithmic, and exponential instructions. For example, the instruction

```
addr r4, r5, fp2
```

adds two real values together and produces an extended-real result.

The following six instructions allow conversion between integers and reals:

<b>cvtir</b>	convert integer to real
<b>cvtilr</b>	convert long integer to long real
<b>cvtri</b>	convert real to integer
<b>cvtril</b>	convert real to long integer
<b>cvtzri</b>	convert truncated real to integer
<b>cvtzril</b>	convert truncated real to long integer

Both the **cvtir** and **cvtilr** instructions can be used to convert an integer to an extended-real value by specifying that the result be placed in a floating-point register.

The convert real-to-integer instructions round off the real value to the nearest integer or long-integer value. For the **cvtri** and **cvtril** instructions, the rounding mode determines the direction the real number is rounded. For the convert truncated real-to-integer instructions (**cvtzri** and **cvtzril**), rounding is always toward zero. The latter two instructions are provided to allow efficient implementation of FORTRAN-like truncation semantics.

Extended-real values can be converted to integers by using a floating-point register as a source operand in either of the convert real-to-integer instructions.

Converting long-real values to integers requires two instructions, as in the following example:

```
movrl g6, fp3
cvtzri fp3, g6
```

The first instruction moves the long-real value to a floating-point register. The second instruction converts the extended-real value to an integer.

## Basic Arithmetic

The following instructions perform the basic arithmetic operations specified in the IEEE standard:

<b>addr</b>	add real
<b>addrl</b>	add long real
<b>subr</b>	subtract real
<b>subrl</b>	subtract long real
<b>mulr</b>	multiply real
<b>mulrl</b>	multiply long real
<b>divr</b>	divide real
<b>divrl</b>	divide long real
<b>remr</b>	remainder real
<b>remrl</b>	remainder long real
<b>roundr</b>	round real
<b>roundrl</b>	round long real
<b>sqrtr</b>	square root real
<b>sqrtrl</b>	square root long real

The round instructions round the floating-point operand to its nearest integral (i.e., integer) value, based on the current rounding mode. These instructions perform a function similar to the convert real-to-integer instructions except that the result is in floating-point format.

## Comparison, Branching, and Classification

Comparison of floating-point values differs from comparison of integers or ordinals because with floating-point values there are four, rather than the usual three, mutually exclusive relationships: less than, equal to, greater than, and unordered.

The unordered relationship is true when at least one of the two values being compared is a NaN. This additional relationship is required because, by definition, NaNs are not numbers, so they cannot have greater than, equal, or less than relationships with other floating-point values.

The following instructions are provided for comparing floating-point values:

<b>cmpr</b>	compare real
<b>cmprl</b>	compare long real
<b>cmpor</b>	compare ordered real
<b>cmporl</b>	compare ordered long real

All of these instructions set the condition code flags in the arithmetic controls to indicate the results of the comparison. With the compare instructions (**cmpr** and **cmprl**), the condition code flags are set to 000<sub>2</sub> for the unordered condition. With the compare ordered instructions (**cmpor** and **cmporl**), the condition code flags are set to 000<sub>2</sub> and an invalid-operation exception is signaled for the unordered condition.

Two branch instructions (**bo** and **bno**) allow conditional branching to be performed on an ordered or unordered condition, respectively. With these instructions, the processor checks the condition code flags for unordered (000<sub>2</sub>) or ordered (111<sub>2</sub>) and branches accordingly.

The classify-real instructions (**classr** and **classrl**) provide a means of determining the class of a floating-point value (i.e., zero, denormalized finite, normalized finite,  $\infty$ , SNaN, or QNaN). The result of this operation is stored in the arithmetic status field of the arithmetic controls.

## Trigonometric

The following instructions provide four common trigonometric functions:

<b>sinc</b>	sine real
<b>sincrl</b>	sine long real
<b>cosr</b>	cosine real
<b>cosrl</b>	cosine long real
<b>tanr</b>	tangent real
<b>tanrl</b>	tangent long real
<b>atanr</b>	arctangent real
<b>atanrl</b>	arctangent long real

The arctangent instructions facilitate conversion from rectangular to polar coordinates.

## Pi

The processor uses the following value for  $\pi$  in its computations:

$$\pi = 0.f * 2^e$$

where:

$$f = \text{C90FDAA2 2168C234 C}_{16}$$

$$e = 2 \text{ if significand is } 0.f$$

(The spaces in the fraction above indicate 32-bit boundaries.)

This value has a 66-bit mantissa, which is 2 bits more than is allowed in the significand of an extended-real value. (Since 66 bits is not an even number of hex digits, two additional zeros have been added to the value so that it can be represented in a hexadecimal format. The least-significant hex digit ( $C_{16}$ ) is thus  $1100_2$ , where the two least significant bits represent bits 67 and 68 of the mantissa.)

If the results of computations that explicitly use  $\pi$  are to be used in the sine, cosine, or tangent instructions, the full 66-bit fraction for  $\pi$  should be used. This insures that the results are consistent with the argument-reduction algorithms that these instructions use. Using a rounded version of  $\pi$  can cause inaccuracies in result values, which if propagated through several calculations, might result in meaningless results.

A common method of representing the full 66-bit fraction of  $\pi$  is to separate the value into two numbers. For example, the following two long-real values added together give the value for  $\pi$  shown above with the full 66-bit fraction:

$$\pi = \text{high}\pi + \text{low}\pi$$

where:

$$\text{high}\pi = 400921\text{FB } 54400000_{16}$$

$$\text{low}\pi = 3\text{DD0B461 } 1\text{A600000}_{16}$$

Here *high* $\pi$  gives the most significant 33 bits of  $\pi$  and *low* $\pi$  gives the least significant 33 bits. Similar versions of  $\pi$  can also be written in the extended-real format.

When using this two-part  $\pi$  value in an algorithm, parallel computations should be performed on each part, with the results kept separate. When all the computations are complete, the two results can be added together to form the final result.

## Logarithmic, Exponential, and Scale

The following instructions provide three different logarithmic functions, an exponential function, and a scale function:

<b>logbnr</b>	log binary real
<b>logbnrl</b>	log binary long real
<b>logr</b>	log real
<b>logrl</b>	log long real
<b>logepr</b>	log epsilon real
<b>logeprl</b>	log epsilon long real
<b>expr</b>	exponent real
<b>exprl</b>	exponent long real
<b>scaler</b>	scale real
<b>scalerl</b>	scale long real

These instructions are described in detail in Chapter 17. The following is a brief description of their functions.

The log binary instructions compute the IEEE recommended function  $\log_b(X)$ . The result is an integral value that is the binary log of  $X$ .

The log instructions compute the function  $Y * \log(X)$ , where the log of  $X$  is the base-2 logarithm.

The log epsilon instructions compute the function  $Y * \log(X + 1)$ , where the log of  $X + 1$  is a base-2 logarithm.

The exponent instructions compute the value  $2^X - 1$ .

The scale instructions perform a multiplication of a floating-point value by a power of 2.

### Arithmetic Versus Nonarithmetic Instructions

The floating-point instructions can be divided into two groups: arithmetic and nonarithmetic. Arithmetic instructions are those that are sensitive to real values, meaning that they distinguish among NaN,  $\infty$ , normalized finite, denormalized finite, and zero values.

All but five of the floating-point instructions are arithmetic. The five nonarithmetic instructions are move-real extended (**movre**), copy-sign real extended (**cpysre**), copy-reversed-sign real extended (**cpysrre**), and classify real (**classr** and **classrl**). These nonarithmetic instructions are insensitive to real values and cannot generate floating-point exceptions or faults.

This distinction between arithmetic and nonarithmetic instructions is important because floating-point exceptions and faults can be signaled only during the execution of arithmetic instructions.

### OPERATIONS ON NANS

As was described earlier in this chapter, the processor supports two types of NaNs: QNaN and SNaN. An SNaN is any NaN value with its most-significant fraction bit set to 0 and at least one other fraction bit set to 1. (If all the fraction bits are set to 0, the value is an  $\infty$ .) A QNaN is any NaN value with the most-significant fraction bit set to 1. The sign bit of a NaN is not interpreted.

In general, when a QNaN is used in one or more arithmetic floating-point instructions, it is allowed to propagate through a computation. An SNaN on the other hand causes a floating invalid-operation exception to be signaled.

The floating invalid-operation exception has a flag and a mask bit associated with it in the arithmetic controls. The mask bit determines how the processor handles an SNaN value. If the floating invalid-operation mask bit is set, the SNaN is converted to a QNaN by setting the most significant fraction bit of the value to a 1. The result is then stored in the destination and the floating invalid-operation flag is set. If the invalid operation mask is clear, a floating invalid-operation fault is signaled and no result is stored in the destination.

When the result is a QNaN, the format of the result is as shown in Table 7-8, depending on the form of the source operands.

Table 7-8. Format of QNaN Results

Source Operands	QNaN Result
Only one operand is NaN, destination is same width	QNaN version of NaN source
Only one operand is NaN, destination is longer	QNaN version of NaN source, with fraction extended with zeros
Only one operand is NaN, destination is shorter	QNaN version of NaN source, with fraction truncated
Both operands are NaNs	QNaN version of source whose fraction field has greatest magnitude, with fraction extended or truncated as described above

In some cases, a QNaN result is returned when none of the source operands are NaNs. Here, a standard QNaN is returned. The significand for the standard QNaN is as follows:

1.1000...00

(For real and long-real destinations, the integer bit will be an implied 1.)

Other than the rules specified above, software is free to use the other bits of a NaN for any purpose.

## EXCEPTIONS AND FAULT HANDLING

Occasionally, a floating-point instruction can result in an exception being signaled. The processor recognizes six floating-point exceptions:

- Floating Reserved Encoding
- Floating Invalid Operation
- Floating Zero Divide
- Floating Overflow
- Floating Underflow
- Floating Inexact

These exceptions can be divided into two categories:

1. Situations in which one or more source operands are inappropriate for an operation and would cause an exception to be signaled.
2. Situations in which the result of an operation is exceptional.

The reserved encoding, invalid operation, and division-by-zero exceptions fall in the first category; the overflow, underflow, and inexact exceptions fall in the second category.



Except for the floating reserved-encoding exception, each of these exceptions has a flag and a mask bit associated with it in the arithmetic controls. When an exception condition occurs, the processor performs one of the following operations:

- If the mask bit for the exception is set, the flag for the exception is set and instruction execution continues, substituting a default value in place of the result.
- If the mask bit for the exception is clear, the flag for the exception is not set and a floating-point arithmetic fault is raised. The processor then stores diagnostic information in the fault information area and diverts instruction execution to a fault handler.

Since the floating reserved-encoding exception does not have a flag or mask bit, it always results in a fault.

#### NOTE

The floating-point exception flags are "sticky," which means that the processor does not implicitly clear them while carrying out floating-point operations. They may be cleared by software.

### Fault Handler

As is described in Chapter 12, when a floating-point fault is signaled, the processor calls a single fault handler. This fault handler determines how to handle the specific fault subtype by interpreting the floating-point exception flags and the information in the fault record.

### Floating-Reserved-Encoding Exception

A reserved-encoding exception occurs as a result of either of the following two conditions:

- When a reserved encoding is used as an operand in a floating-point instruction, or
- When a denormalized value is used as an operand in a floating-point instruction and the normalizing-mode bit in the arithmetic controls is clear.

The first condition is rare. It can only occur if a program presents an extended-real value to the processor that has a zero j-bit (integer part) and a non-zero biased exponent.

The second condition was discussed earlier in this chapter in the section titled "Normalizing Mode." This condition is also rare, since the vast majority of programs run with the normalizing mode enabled.

There is neither a flag nor a mask bit for this exception. When a reserved-encoding exception occurs, the processor raises a floating-reserved-encoding fault and does not store a result.

### Floating-Invalid-Operation Exception

The invalid-operation exception indicates that one of the source operands is inappropriate for the type of operation being performed. The following conditions cause this exception to be signaled:

- Any arithmetic operation on an SNaN
- Addition of infinities of unlike sign
- Subtraction of infinities of like sign
- Multiplication of zero by  $\infty$
- Division of zero by zero or  $\infty$  by  $\infty$
- Remainder of  $x$  by  $y$ , if  $y$  is zero or  $x$  is  $\infty$
- Square root of a negative, nonzero value
- Conversion of a NaN from floating-point format to integer format
- Sine, cosine, or tangent of  $\infty$
- $y * \log(x)$ , if:
  - $x$  is negative and nonzero,
  - $y$  is zero and  $x$  is  $\infty$ ,
  - $y$  and  $x$  are zero, or
  - $y$  is  $\infty$  and  $x$  is 1
- Log epsilon of  $(y, x)$ , if  $y$  is  $\infty$  and  $x$  is 0
- Compare ordered, if a source operand is a NaN

When a floating-invalid-operation exception occurs and its mask is set, the following occurs:

- When the result is a floating-point value, the standard QNaN value is stored in the destination and the floating-invalid-operation flag is set. (A discussion of how the processor handles NaNs was provided earlier in this chapter in the section titled "Operations on NaNs.")
- When the result is an integer, the maximum negative integer is stored in the destination and the floating-invalid-operation flag is set.

When the mask is clear, no result is stored; the floating-invalid-operation flag is not set; and the floating-invalid-operation fault is signaled.

### Floating-Zero-Divide Exception

The floating-zero-divide exception is signaled when an exact non-finite result would be produced from finite operands. (Note that a different exception, overflow, is signaled when an infinite result is produced inexactly from finite operands.) The most common example of this exception is a division operation, where the divisor is zero and the dividend is a nonzero, finite value.

When the floating-zero-divide mask is set: a correctly signed  $\infty$  is stored in the destination and the floating-zero-divide flag is set. When the mask is clear, no result is stored; the floating-zero-divide flag is not set; and a floating-zero-divide fault is signaled.

### Floating-Overflow Exception

The overflow exception occurs when the infinitely precise result of a floating-point instruction exceeds the largest allowable finite value for the specified destination format. For example, if the destination format is real (32 bits), overflow occurs when the infinitely precise result falls outside the range  $-1.0 * 2^{128}$  to  $1.0 * 2^{128}$  (exclusive), where 128 is the unbiased exponent of the result. For long-real (64 bits) values, the overflow threshold range is  $-1.0 * 2^{1024}$  to  $1.0 * 2^{1024}$ ; for extended-real (80 bits) values, it is  $-1.0 * 2^{16384}$  to  $1.0 * 2^{16384}$ .

When the floating-overflow mask is set, a rounded result is stored in the destination and the floating-overflow flag is set. The current rounding mode determines the method used to round the result.

When the mask is clear: no result is stored in the destination and the floating-overflow flag is not set. Instead, the processor stores the result in extended-real format in the fault information area. The fraction of the extended-real value is rounded to the instruction's destination precision. For example, if the destination operand's format is real (32 bits), the extended-real fraction is rounded to 23 bits, with the 40 least-significant bits filled with zeros.

If the exponent exceeds the range of the extended-real format (16383 unbiased), then the exponent is divided by  $2^{24576}$  and a flag (bit 1 of the fault flags byte or override flags byte) is set in the fault information area to indicate that the exponent has been bias adjusted. After this fault information is stored, a floating-overflow fault is signaled.

When using the scale instructions (**scaler** or **scalerl**), massive overflow can occur, where the infinitely precise result is too large to be represented, even with a bias-adjusted exponent. Here, a properly signed  $\infty$  is stored in the fault record.

The floating-overflow exception cannot occur on a conversion from floating-point format to integer format (although an integer overflow exception can occur).

### Floating-Underflow Exception

An underflow condition occurs when the infinitely precise result of a floating-point instruction is less than the smallest possible normalized, finite value for the specified destination format. For example, for the real (32-bit) format, underflow occurs when an infinitely precise result falls in the range  $-1.0 * 2^{-126}$  to  $1.0 * 2^{-126}$  (exclusive), where -126 is the unbiased exponent. For long-real (64 bits) values, the underflow threshold range is  $-1.0 * 2^{1022}$  to  $1.0 * 2^{1022}$ ; for extended-real (80 bits) values, it is  $-1.0 * 2^{16382}$  to  $1.0 * 2^{16382}$ .

When a floating-underflow condition occurs, the setting of the floating-underflow mask determines how the processor handles the condition.

If the mask is set when an underflow condition occurs, the processor goes ahead and denormalizes the result. Then if the result is exact, it is stored in the destination and the floating-underflow exception is not signaled, nor is the floating-underflow flag set. If, on the other hand, the denormalized result is inexact, the floating-underflow flag is set and the processor goes on to handle the inexact condition as described in the next section.

If the floating-underflow mask is clear when an underflow condition occurs, no result is stored in the destination and the floating-underflow flag is not set. Instead, the processor stores the result in extended-real format in the fault information area, with the fraction of the extended-real value rounded to the instruction's destination precision. For example, if the destination precision is real (23-bit fraction), the 40 least-significant bits of the fraction are set to 0.

If the exponent of the value stored is less than the minimum allowable value in the extended-real format ( $-16,382$  unbiased), then the exponent is multiplied by  $2^{24576}$  and a flag (bit 1 of the fault or override flags byte) is set in the fault information area to indicate that the exponent has been bias adjusted. After this information is stored, a floating-underflow fault is signaled.

The scale instructions can cause massive underflow to occur, where the infinitely precise result is too small to be represented, even with a bias-adjusted exponent. Here, a properly signed zero is stored in the fault record.

Refer to the section later in this chapter titled "Floating-Point Underflow Condition" for more information on the interaction of the floating underflow and inexact exceptions.

### Floating-Inexact Exception

The floating-inexact exception occurs when an infinitely precise result cannot be encoded in the format specified for the destination operand. Either of the following two conditions can cause an inexact exception to be signaled:

- When a result is rounded and the result is not exact
- When overflow occurs and the floating-overflow mask is set

If the floating-inexact mask is set when an inexact condition occurs and an unmasked overflow or underflow condition does not occur, the rounded result is stored in the destination and the floating-inexact flag is set. The current rounding mode determines the method used to round the result.

If the floating-inexact mask is clear when an inexact condition occurs, the floating-inexact flag is not set and one of the following operations is carried out:

- If only the inexact condition has occurred, the processor stores the rounded result in the specified destination, then raises a floating-inexact fault.
- If the inexact condition occurs along with overflow or underflow, no result is stored in the destination. Instead, the processor stores the result in extended-real format in the fault information area, as described for the floating overflow and underflow exceptions, then raises a floating-inexact fault.

Refer to the following section for more information on the interaction of the floating underflow and inexact exceptions.

### Floating-Point-Underflow Condition

Two aspects of underflow are important in numeric processing: the "tininess" of a number and "loss of accuracy." A result is tiny when it is nonzero and its exponent is between  $\pm 2^{E_{\min}}$ , where  $E_{\min}$  is the smallest unbiased exponent allowed in the destination format. For example, if the destination format is long-real (64-bit format), a result is tiny if it is nonzero and in the range of  $+1 * 2^{-1022}$  to  $-1 * 2^{-1022}$ . The ability to detect a tiny result is important because such a result may cause an exception to be signaled in a later operation (e.g., overflow on a division).

Loss of accuracy occurs when a tiny result is approximated as part of the denormalization process so that it will fit into the destination format.

In the 80960MC processor, tininess is detected after rounding as an underflow condition. Loss of accuracy is detected as an inexact condition.

The algorithm in Figure 7-6 shows how the processor responds to these two conditions, when a floating-point operation produces a tiny result.

An important point to note in this algorithm is that if the underflow mask is set, an underflow exception is signaled only if the denormalized result is inexact. If the denormalized number is exact, no flags are set and no faults are signaled.

The floating-inexact exception occurs when an infinitely precise result cannot be encoded in the format specified for the destination operand. Either of the following two conditions can cause an inexact exception to be signaled:

- When a result is rounded and the result is not exact
- When overflow occurs and the floating-overflow mask is set

If the floating-inexact mask is set when an inexact condition occurs and an unmasked overflow or underflow condition does not occur, the rounded result is stored in the destination and the floating-inexact flag is set. The current rounding mode determines the method used to round the result.

If the floating-inexact mask is clear when an inexact condition occurs, the floating-inexact flag is not set and one of the following operations is carried out:

- If only the inexact condition has occurred, the processor stores the rounded result in the specified destination, then raises a floating-inexact fault.
- If the inexact condition occurs along with overflow or underflow, no result is stored in the destination. Instead, the processor stores the result in extended-real format in the fault information area, as described for the floating overflow and underflow exceptions, then raises a floating-inexact fault.

```
generate infinitely precise result # exponent and significand;
if exponent < underflow threshold
  then
    if underflow fault mask clear
      then
        goto underflow fault handler;
        exit algorithm;
      else generate denormalized number
        if denormalized significand equals infinitely precise significand
          then
            store denormalized result in destination;
            # no underflow is signaled;
          else
            set underflow flag in AC;
            if inexact fault mask is clear
              then
                goto inexact fault handler;
                exit algorithm;
              else
                set inexact flag in AC;
                store denormalized result in destination;
              end if;
            end if;
          end if;
        else
          if infinitely precise result is inexact
            then
              if inexact fault mask is clear
                then
                  goto inexact fault handler;
                  exit algorithm;
                else
                  set inexact flag in AC;
                  store normalized result in destination;
                end if;
              else
                store normalized result in destination;
              end if;
            end if;
          end if;
        exit algorithm
```

Figure 7-6: Interaction of Floating Underflow and Inexact Exceptions





---

---

## CHAPTER 8 MEMORY MANAGEMENT

This chapter describes the 80960MC processor's memory management facilities. Included is a discussion of the physical memory requirements, physical addressing, and the virtual-memory-management system. The information presented here should be of interest only to operating-system designers, particularly those designing the virtual-memory-management mechanism for the operating-system kernel. Application programmers and compiler writers may skip this chapter.

### INTRODUCTION

A major feature of the 80960MC processor is its virtual-memory-management facilities. These facilities support a conventional demand-paged, virtual-memory system, in which 4K-byte pages of virtual memory are mapped to physical memory. This general purpose system can be used in any of the follow applications:

- In a single-process system to map a large virtual address space into a smaller physical address space.
- In a multitasking system to provide each process with a separate address space.
- In a multiprocessing system to provide a means for multiple processors to share a common memory.

The processor's virtual-memory-management facilities consists of a set of memory-management data structures and on-chip address translation capabilities. Once the operating system has set up these data structures, the processor provides automatic translation of virtual addresses into physical addresses.

The majority of this chapter is devoted to a discussion of the virtual-memory system. If the processor is going to be used strictly in the physical-addressing mode, only the first sections of this chapter, which describe the physical address space and physical memory requirements, need to be read.

### PHYSICAL-ADDRESSING MODE VERSUS VIRTUAL-ADDRESSING MODE

The 80960MC processor provides two address-interpretation modes: physical-addressing mode and virtual-addressing mode. When operating in physical-addressing mode, the processor interprets each address operand in an instruction as a physical address and sends the address out to the bus unchanged.

In virtual-addressing mode, the processor interprets each address operand as a virtual address. An on-chip memory management unit (MMU) translates the virtual address into a physical address, which the processor then sends out to the bus.

The addressing mode flag in the processor controls determines which addressing mode the processor is operating in. When this flag is clear, the processor operates in physical-addressing mode; when the flag is set, the processor operates in virtual-addressing mode.

## PHYSICAL MEMORY

The processor can address a physical address space of up to  $2^{32}$  bytes. This address space can be mapped to read-write memory, read-only memory, and memory-mapped I/O.

The physical address space is linear (or flat): there are no subdivisions of the address space such as segments. For the purpose of memory management, the kernel may subdivide physical memory into pages. But from the point of view of the processor, the physical address space is linear.

All of the physical address space is available for general use except the upper 16M bytes ( $\text{FF000000}_{16}$  to  $\text{FFFFFFFF}_{16}$ ), which are reserved for special functions. (These functions are described in Chapter 11.)

A physical address is a 32-bit value in the range 0 to  $\text{FFFFFFFF}_{16}$ . A physical address can be used to reference a single byte, 2 bytes, 4 bytes, 8 bytes, 12 bytes or 16 bytes of memory depending on the instruction being used. (Refer to the descriptions of the load and store instructions in Chapter 17 for information on multiple-byte addressing.)

### Physical-Memory Restrictions

The processor requires that the physical memory that it accesses has the following capabilities:

- It must be byte addressable.
- It must support burst transfers (i.e., transfers of blocks of contiguous bytes up to 16 bytes in length).
- It must guarantee *indivisible* access (read or write) for memory addresses that fall within 16-byte boundaries.
- It must guarantee *atomic* access for memory addresses that fall within 16-byte boundaries.

The latter two capabilities are required to allow multiple processors to share a common physical address space conveniently.

An indivisible access guarantees that a processor reading or writing a set of memory locations will complete the operation before another processor can read or write the same location. The processor requires indivisible access within an aligned, 16-byte block of memory.

An atomic access is a read-modify-write operation. Here the memory controller guarantees that once a processor begins a read-modify-write operation on a set of memory locations, it is allowed to complete the operation before another processor is allowed to access the same location.

As described above, the processor requires that when one processor is performing an atomic operation within an aligned, 16-byte block, other processors are delayed from performing another atomic operation within that block until the first operation has been completed.

The 80960MC processor provides two features to aid in implementing the requirements of physical memory described above: SIZE lines and a LOCK line on the local bus.

The SIZE lines indicate the length of a memory access in bytes. These lines can be used to specify 1-, 2-, 4-, 8-, 12-, or 16-byte lengths. When making a multiple-byte access, the processor thus sends the memory controller a base address, on the address lines, and a length, on the SIZE lines.

The LOCK line is used to synchronize atomic operations. When a processor performs an atomic operation, it first examines the LOCK line. If it is asserted, the processor waits until the line is not asserted (i.e., spins on the LOCK line). If the line is not asserted, the processor asserts the LOCK line when it is performing an atomic read and deasserts the line when it performs the companion atomic write.

For systems that use only the processor's local bus, the LOCK line mechanism allows only one atomic operation to be carried out in memory at one time. For larger systems that use the Intel advanced processor bus (AP Bus), the Bus Extension Unit (BXU) component allows multiple processors on the bus to execute several atomic operations at once on different blocks of memory. Refer to the *80960MC Hardware Designer's Reference Manual* for detailed information on atomic operations.

### Caching of Memory Accesses

The processor supports caching of memory accesses. Caching allows a memory access to be delayed (e.g., write back) or grouped with contiguous memory accesses to form a single memory transaction (e.g., cache fill).

The processor does not perform the caching function; however, it does provide a means of informing a cache manager whether or not a memory access is "cacheable."

When operating in the physical-address mode, all memory accesses are considered cacheable.

### VIRTUAL-MEMORY-MANAGEMENT SYSTEM

The processor's virtual-memory-management system is designed to perform the following functions:

- Allow the mapping of a large, virtual address space into a smaller physical address space using 1- or 2-level page tables.
- Provide a convenient means of managing multiple process address spaces in multitasking operating systems.
- Provide a method of addressing architecture-defined data structures.

The first function is handled by means of a traditional paging mechanism that uses page tables and optional page-table directories to map the virtual address space into physical address space in 4K-byte pages.

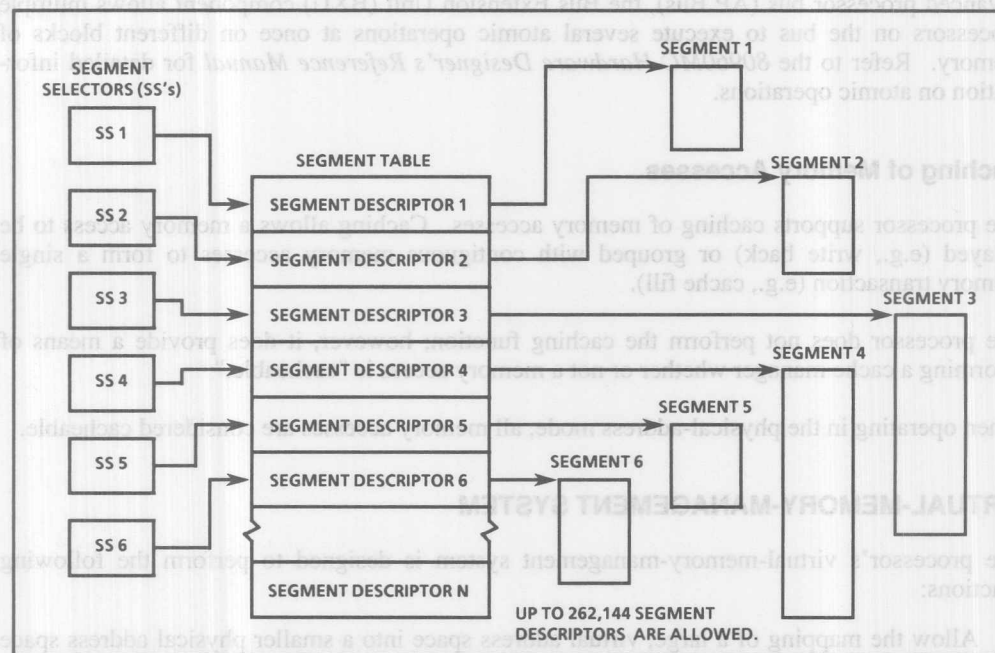
The second and third functions are handled through a central table, called the *segment table*, which the processor uses to locate a specific address space or system data-structure in physical memory.



The following discussion first presents the concept of the segment table and the mechanism used to implement this concept. Then, the paging mechanism is described. Finally, the method the operating system uses to set up and maintain these memory management structures is given.

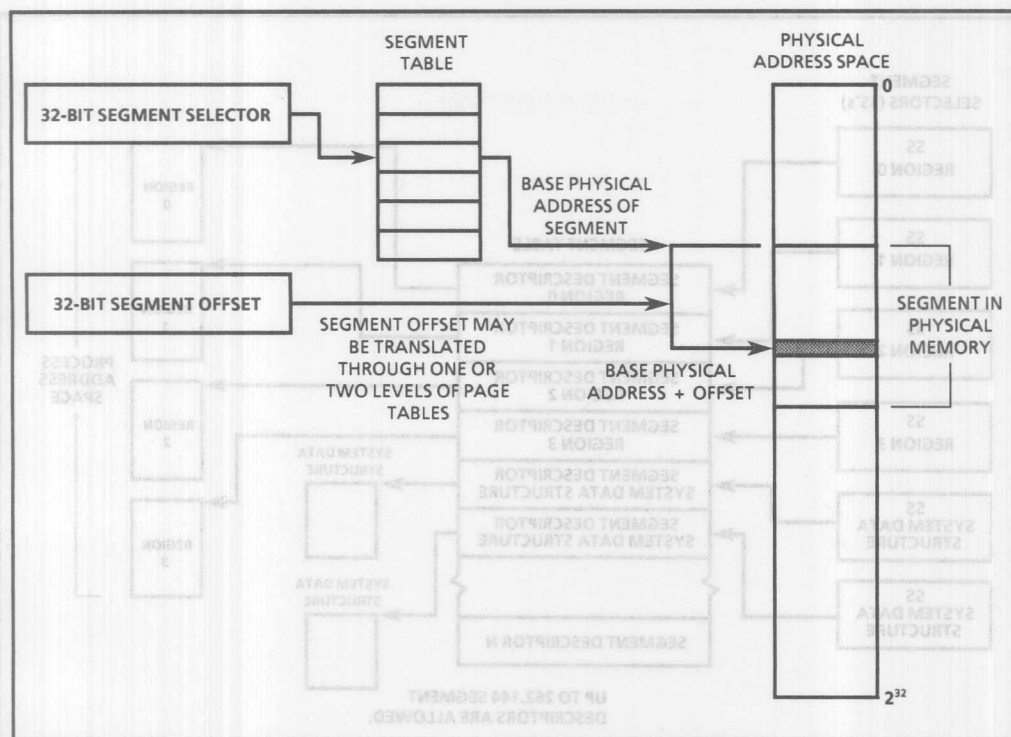
## SEGMENT-TABLE OVERVIEW

The segment table is a data structure that resides in physical memory. This table provides the processor with a system-wide addressing mechanism, which allows the processor to locate all the process address spaces and system data structures that the kernel has created. It also allows many process address spaces and data structures to be mapped into physical memory at one time. Figure 8-1 shows a conceptual view of the segment table.



**Figure 8-1: Conceptual View of the Segment Table**

The segment table is made up of a collection of *segment descriptors*. Each segment descriptor points to an individual *segment*. A segment is defined as a contiguous address space of from  $16$  to  $2^{32} - 1$  bytes. Figure 8-2 shows a segment and the mechanism used to address a byte in a segment.



**Figure 8-2: Segment Addressing**

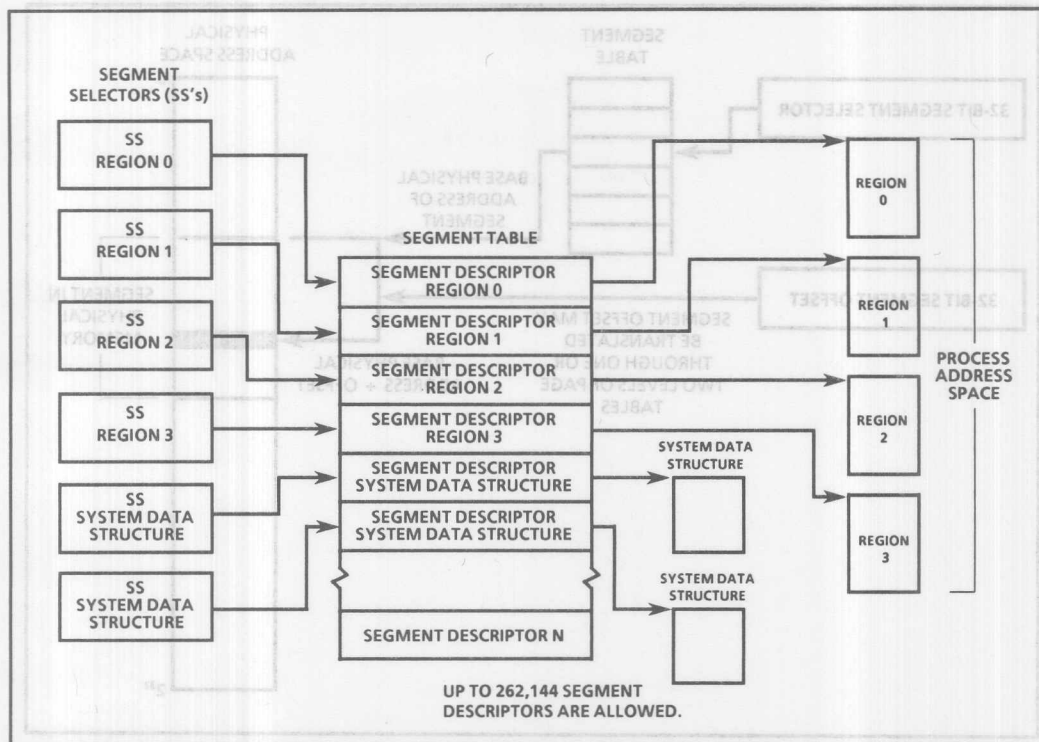
A segment is addressed by means of a 32-bit data structure called a *segment selector* (SS). An SS contains an index into the segment table to the location of the segment descriptor for the segment. When the operating system creates a segment, it assigns a unique SS to the segment.

To locate a byte in a segment, the processor then needs two items: the SS for the segment and a 32-bit offset into the segment. The processor uses the SS to locate the segment descriptor for the segment in the segment table. From this segment descriptor, it gets the physical address of the base (first byte) of the segment. It then uses the offset to locate the selected byte in the segment.

When paging is used, the offset is translated through page tables and an optional page table directory to get the physical address of the selected byte in the segment.

## USES OF SEGMENTS

The processor uses segments in two ways, as shown in Figure 8-3. The first way is as a means of addressing the four regions that make up the address space for a process.



**Figure 8-3: Uses of Segments**

As was described in Chapter 3, part of the execution environment for the processor is the address space, which can range from 1 to  $2^{32}$  bytes. When using the processor's virtual-memory system, the address space is divided into four regions. Each of these regions is contained in a segment. To access the address space, the processor must have four SS's, one for each region.

In a multitasking system, each process is assigned its own address space. Each process address space is made up of four regions, which the processor locates with four SS's.

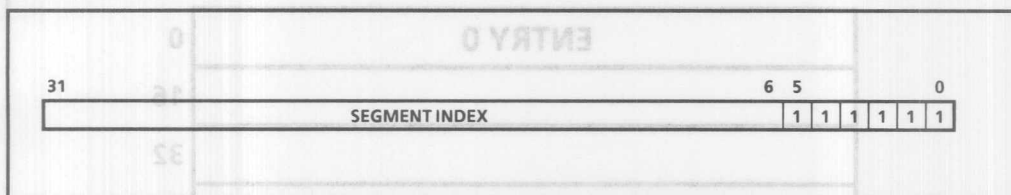
The second way that the processor uses segments is to address system data structures. The processor defines several system data structures such as the PCB and the system procedure table. Each of these data structures is contained in a segment. The processor is able to access data in these data structures by means of the SS for the segment that contains the data structure.

## SEGMENT-TABLE DATA STRUCTURES

The following sections describe the actual structure of an SS, a segment table, and a segment descriptor.

## Segment Selector

Figure 8-4 shows the structure of an SS. The first six bits are always set to 1. Bits 6 through 31 give the entry number of the segment selector in the segment table. (Since segment descriptors are aligned in the segment table on 16-byte boundaries, the segment index actually gives the 26 most significant bits of the offset into the segment table of the first byte of the segment selector. The processor assumes the six least-significant bits are zero.) This structure allows the operating system to create up to  $2^{26}$  unique SS's. However, the largest allowable segment table can have only 262,144 ( $2^{18}$ ) segment descriptors.



**Figure 8-4: Segment Selector**

A segment selector can be stored anywhere in the address space for a process or in specific places in system data structures. They are, however, useful for only two purposes:

- Certain instructions use an SS as an operand. These instructions can only be executed while in the supervisor mode and are thus normally used only by the operating system.
- The processor fetches SS's from various system data structures and uses them to access system management information. For example, the processor gets the SS for region 3 of the process address space from the processor control block.

Applications programs will generally not use SS's.

### NOTE

When the processor uses an SS for its intended purpose (as a pointer to a segment), it expects the 6 least-significant bits of the SS to be set to 1. If they are not, the processor's behavior is unpredictable.

Once the processor uses an SS, however, it clears some of these bits; and, if a program examines an SS that the processor has used, some of these bits may be zero.

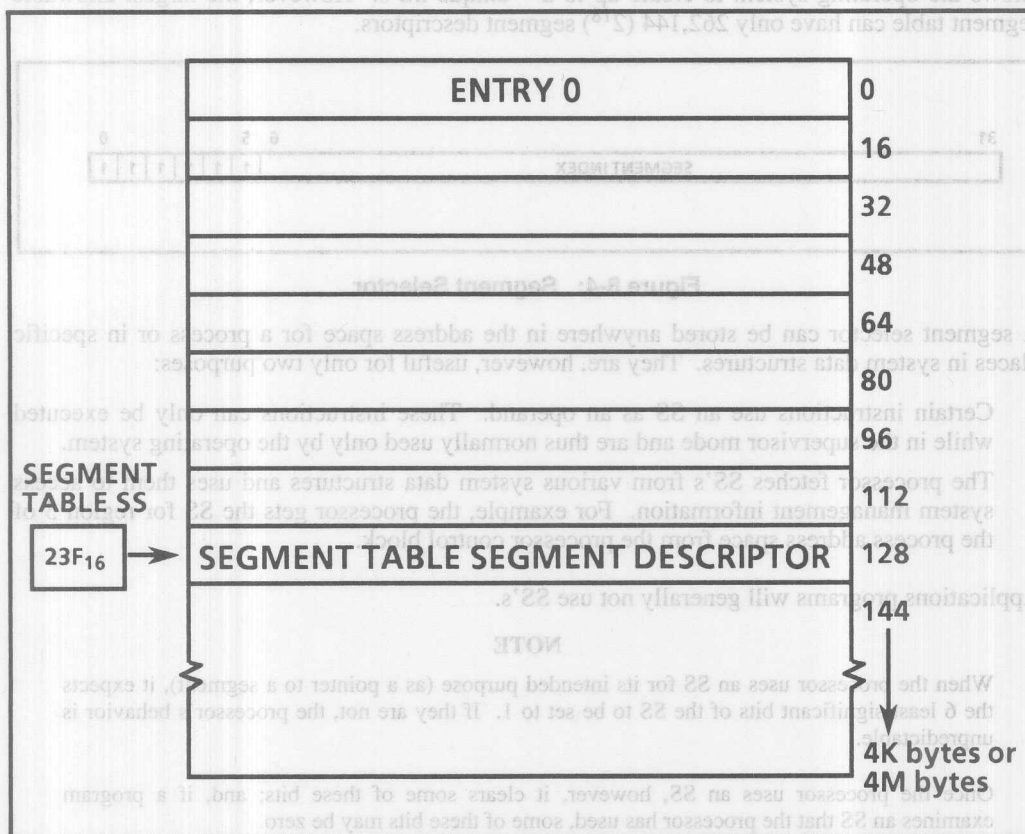
To insure predictable behavior of the processor, it is good programming practice to reset the 6 least-significant bits of the SS to 1 any time a program moves an SS that the processor has already used.

For example, if a program removes an SS for a PCB from a dispatch port, it should set these bits to 1 as a matter of course, before it places the SS in a data structure or instruction where the processor will use the SS for its intended purpose.

## Segment Table

The segment table is itself contained in a segment and has an SS. This allows the processor to locate the segment table in physical memory.

Figure 8-5 shows the structure of a segment table. It is simply a collection of 16-byte segment descriptors, with no header.



**Figure 8-5: Segment Table**

Except for index entry eight (with entry zero being the lowest numbered entry), the segment descriptors can be assigned to any segment. Entry eight is reserved for the segment descriptor for the segment table. The SS for the segment table is thus always 0000023F<sub>16</sub>.

There are two sizes of segment tables: a small segment table and a large segment table. A small segment table is 4096 bytes (1 page) in length and can contain up to 256 segment descriptor entries. A large segment table can be up to 4M bytes in length and can contain up to 262,144 segment descriptors.

## Segment Descriptors

A 16-byte segment descriptor provides mapping information to allow the processor to locate a specific segment in physical memory. It also provides type information and in some cases access information to tell the processor how the segment may be used or how it has been used.

The segment descriptor fields contain the following pieces of information:

- The base physical address of the segment
- The size of the segment
- The access status
- Whether or not the segment is in physical memory
- The paging method
- The segment type

Figure 8-6 shows a generic segment descriptor with the fields labeled. The function of each of these fields is described in the following paragraphs. The entries required in each fields for specific types of segment descriptors (such as, port segment descriptors, process segment descriptors, etc.) are given later in this chapter in the section titled "Segment Types".

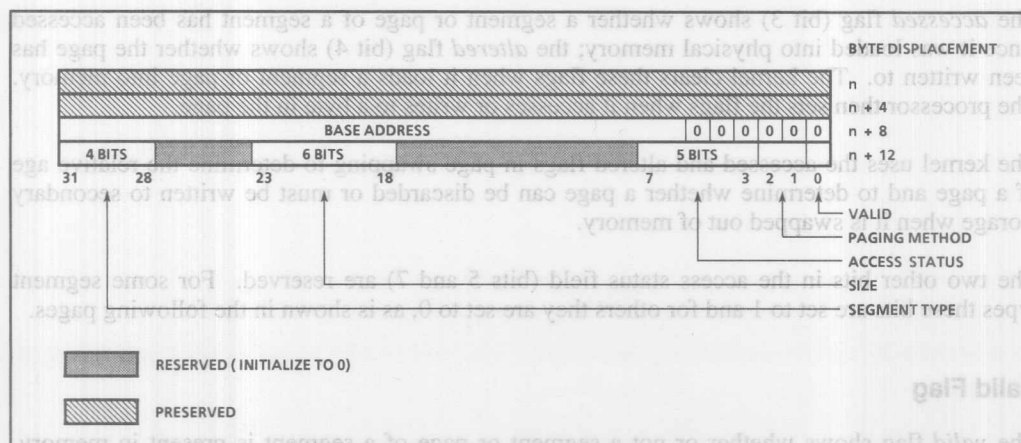


Figure 8-6: Generic Segment Descriptor

### NOTE

The shaded areas in Figure 8-6 and in the following figures indicate reserved and preserved areas of a segment descriptor. Refer to Chapter 1 for an explanation of these terms.

### Base Address

The base address field gives the physical address of byte 0 of the segment being referenced. If the segment is a paged segment, this field gives the base address of a page table or a page-table directory.



## Size

The size field determines the length of the segment according to the following relationship:

$$\text{segment length in bytes} = 64 * (\text{SIZE} + 1)$$

For most segment types, the size field is either not used or the value to be placed in this field is predefined. However, for a few segment types this field is used to determine the size of the segment, as shown later in this chapter.

## Access Status

The three flags in the access status field determine how a segment or page can be used or has been used. The processor and kernel use these flags to facilitate page swapping. For paged segments, some of these flags may not be used at the segment descriptor level. Instead, they are set in the page table or page-table-directory entries.

The *cacheable* flag (bit 6) determines whether or not a segment or page of a segment can be cached. When this flag is set the segment or page is cacheable. Caching of memory accesses was described earlier in this chapter in the section titled "Caching of Memory Accesses."

The *accessed* flag (bit 3) shows whether a segment or page of a segment has been accessed since it was loaded into physical memory; the *altered* flag (bit 4) shows whether the page has been written to. The kernel clears these flags when it loads a segment or page into memory. The processor then sets the flags when it accesses or writes to a byte in the page.

The kernel uses the accessed and altered flags in page swapping to determine the relative age of a page and to determine whether a page can be discarded or must be written to secondary storage when it is swapped out of memory.

The two other bits in the access status field (bits 5 and 7) are reserved. For some segment types these bits are set to 1 and for others they are set to 0, as is shown in the following pages.

## Valid Flag

The *valid* flag shows whether or not a segment or page of a segment is present in memory. When this flag is set, the segment is present; when it is clear, the page is not present. When the processor attempts to access a segment or page, it checks this flag to determine if the segment or page is present. If the valid flag is clear, the processor raises a virtual-memory fault. The fault handler routine then calls upon the kernel to load the segment or page into memory.

When the valid flag is set to 0, the processor does not interpret the other bits in the segment descriptor. Software is then free to use these bits for other purposes. For example, if a segment is not in physical memory, the base address field might be used to store the location of the segment in a mass storage device (such as a disk).

## Paging Method

The paging method field shows whether the segment is unpaged (01), paged (10), or bipaged (11). The value in this field must be as is shown in the following sections for each segment descriptor type.

## Segment Types

The processor recognizes the following nine types of segments:

- Simple Region
- Paged Region
- Bipaged Region
- Process Control Block
- Port
- Procedure Table
- Semaphore
- Small Segment Table
- Large Segment Table

The segment descriptor is set up differently for each segment type, as is described in the following paragraphs. For some of these segment types (but not all), the type is shown in the type field. For those segments types where the type is specified, the processor checks this type field before accessing the rest of the data in the segment descriptor to insure that the segment being accessed is the correct type. In cases where the processor performs type checking on segment descriptors, it signals a type fault if an inappropriate type is found.

The following paragraphs describe what must be placed in each of the segment-descriptor fields, depending on the type of segment that the segment descriptor is pointing to.

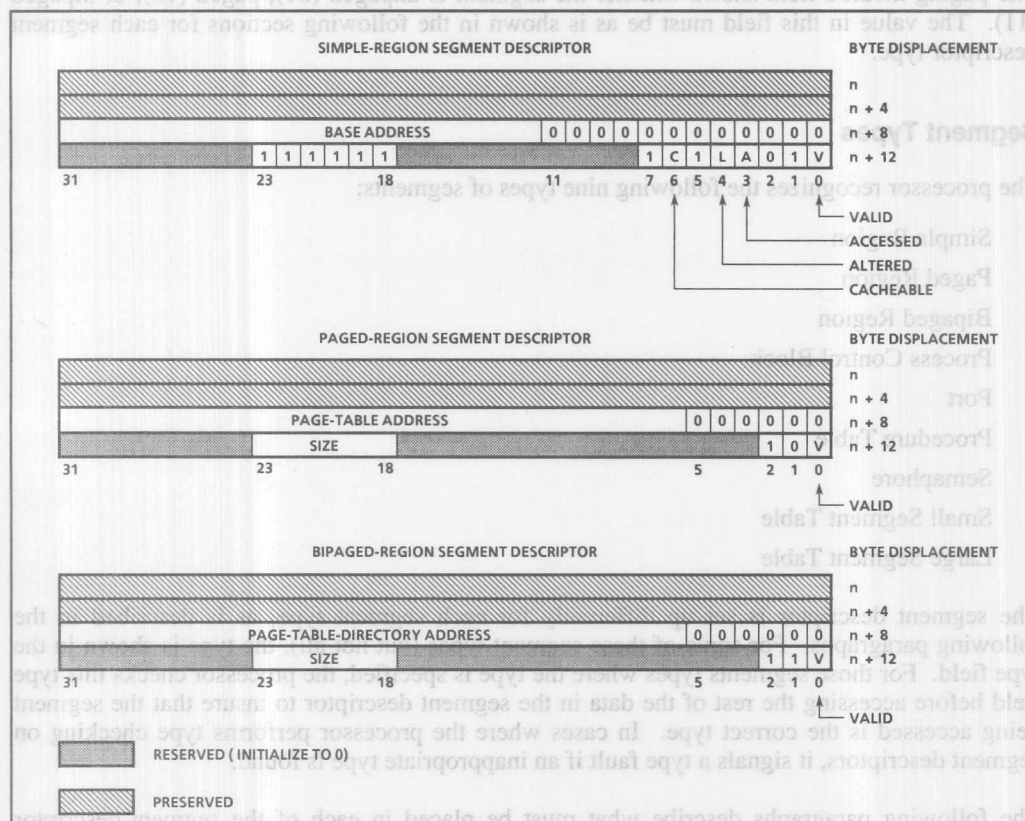
## Region Descriptors

Each region of an address space is contained in a segment. A region segment can be a simple region, a paged region, or a bipaged region. For each of these three types of regions, the segment descriptor is set up slightly different. Figure 8-7 shows the segment descriptors for the three types of regions.

**Simple Region.** A simple region is a one-page segment (4096 bytes) that is mapped into physical memory as a contiguous page.

The base address for a simple region must fall on a page boundary in physical memory, so the 12 least-significant bits of the base address field are set to zero. The size field is set to  $63_{10}$ , indicating 4K bytes length.

Since the simple region descriptor points directly to the segment in memory, the three access flags (accessed, altered, and cacheable) are set and examined by the processor and kernel.



**Figure 8-7: Region Segment Descriptors**

**Paged Region.** A paged region is a segment that is mapped into physical memory by means of a page table. A paged region may be from 4096 bytes to 4096K bytes in length.

The base address field for a paged-region descriptor points to the base physical-address of a page table. This address must fall on a 64-byte boundary, so the 6 least-significant bits of the base address field are set to zero.

A page-table can be up to a page in length as determined by the size field. Each page-table entry is 4 bytes, so the number of entries in the page table is as follows:

$$\text{Number of Page-Table Entries} = 16 * (\text{SIZE} + 1)$$

For a paged region, the access information is stored in the page-table entries. The access status flags in the segment descriptor are thus set to 0 and the valid flag shows whether or not the page table is present in memory.

**Bipaged Region.** A bipaged region is a segment that is mapped into physical memory by means of two levels of page tables. A page-table directory forms the first level. Entries in the page-table directory then point to up to 1024 page tables. A bipaged region may be from 4096 bytes to 4096M bytes in length.

The segment descriptor for a bipaged region is similar to that of a paged region descriptor. The base address field gives the base physical-address of a page-table directory, which must fall on a 64-byte boundary.

A page-table directory can be up to a page in length as determined by the size field. The number of 4-byte entries in the page-table-directory is determined by the same relationship, as is shown above for a page table in a paged region.

As with paged regions, all of the access information except the valid flag is stored in the page-table-directory and page-table entries.

### Process, Port, and Procedure-Table Descriptors

A process-segment descriptor points to a segment that contains a process control block (PCB); a port-segment descriptor points to a segment that contains a dispatch port or a communication port; and a procedure-table segment descriptor points to a segment that contains a procedure table. Figure 8-8 shows the format for each of these types of segment descriptors.

#### NOTE

A PCB and a port are architecture-defined data structures. The PCB is described in Chapter 13; the port is described in Chapter 14.

The formats for these segment descriptors are identical, except that the value in the type field is different for each type of descriptor.

The base address for each of these segments must fall on a 64-byte boundary in physical memory and the segment as a whole must not span a 4096-byte boundary. Spanning a 4096-byte boundary will cause unpredictable results when the segment is accessed.

The sizes of the process and port segments are defined by the PCB and port data structures. The size of the procedure table segment is 1088 bytes.

These segments must always be present in physical memory, so the valid, accessed, and altered flags are always set to 1. The cacheable flag can be set to allow caching of the segment.

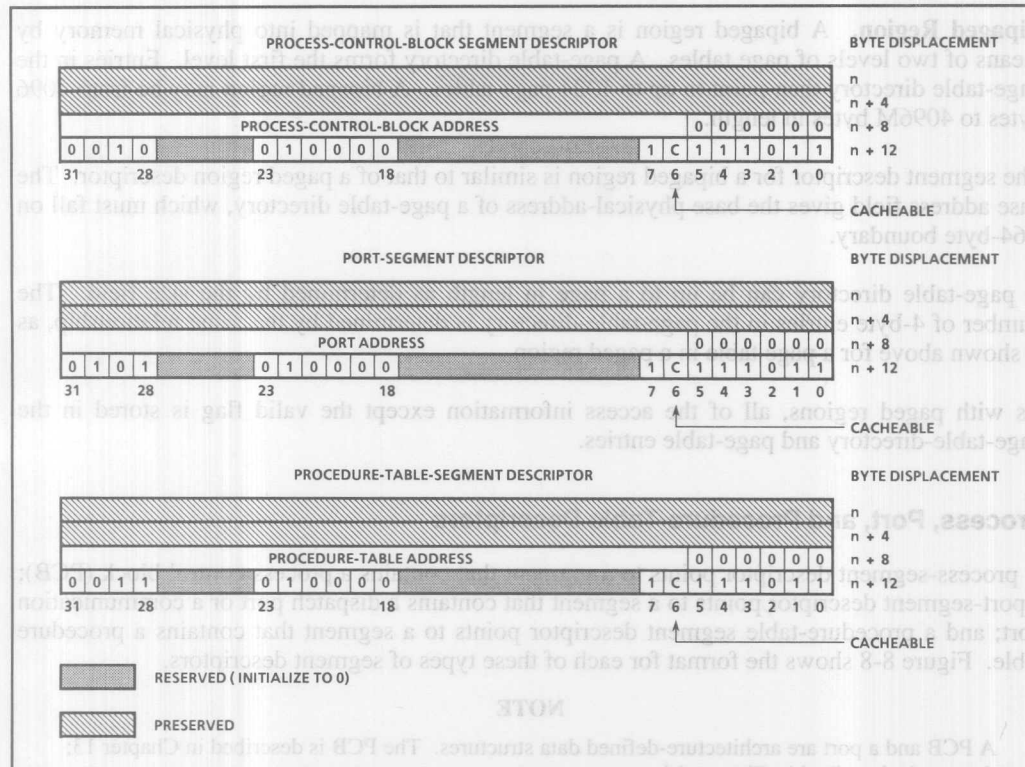


Figure 8-8: Process, Port, and Procedure-Table Segment Descriptors

## Segment-Table Descriptors

Figure 8-9 shows the formats for the two types of segment-table descriptors: one for a small segment table and another for a large segment table.

A small segment table is mapped to a page of physical memory. The base address in the small segment table descriptor must point to a 4096-byte (page) boundary in physical memory. The 12 least-significant bits of the base address are thus set to zero.

A small segment table must always be in physical memory, so the accessed, altered, and valid flags are set to 1. Whether or not a small segment table is cacheable is optional.

A large segment table is mapped to physical memory by means of a page table. The base address in the large segment table descriptor then points to the base address of a page table, which must be located on a page boundary.





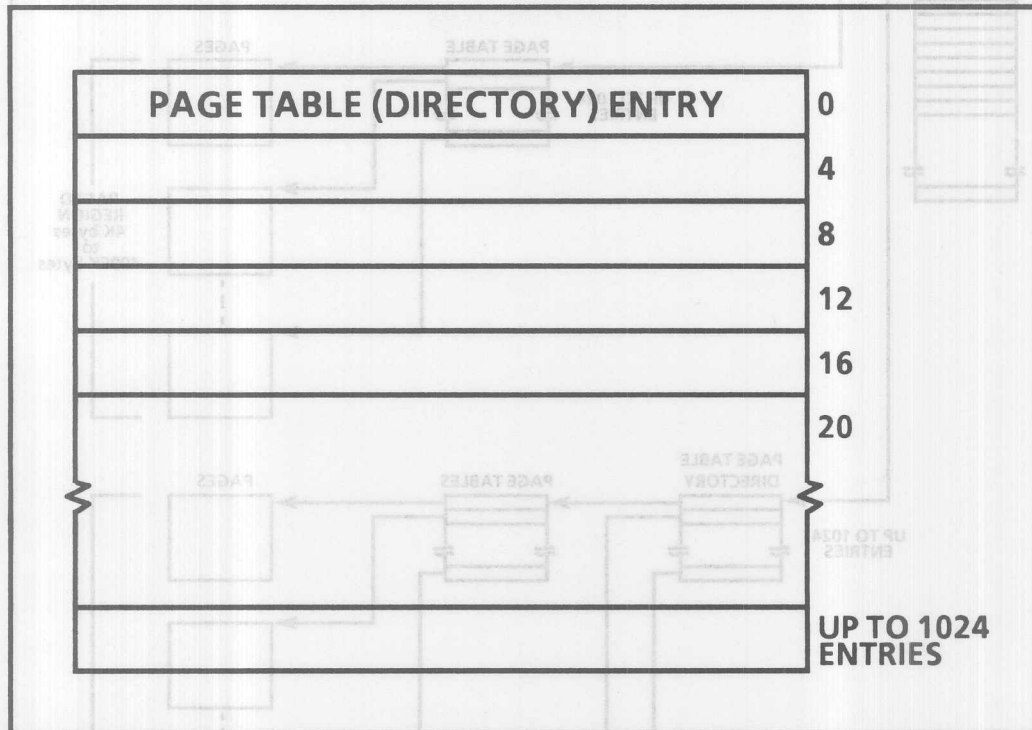






## Page Table and Page-Table-Directory Structure

As is shown in Figure 8-13, page tables and page-table directories are made up of 4-byte entries. (There is no table header.) Both types of tables can be up to one page in length, which allows up to 1024 entries per table.



**Figure 8-13: Page Table or Page-Table-Directory Structure**

One-level paging can be used to page segments of from 4096 bytes to 4096K bytes in length; two-level paging can be used to page segments of from 4096 bytes to 4096M bytes in length.

When using one-level paging, the size field in the paged segment descriptor determines the number of entries in a page table. Likewise, when using two-level paging, the size field in the bipaged segment descriptor determines the number of entries in the page-table directory. However, when setting up a bipaged segment, the page tables that the page-table directory points to have a set length of one page.

## Page Table and Page-Table-Directory Entries

Figure 8-14 shows the structure of the page table and page-table-directory entries.

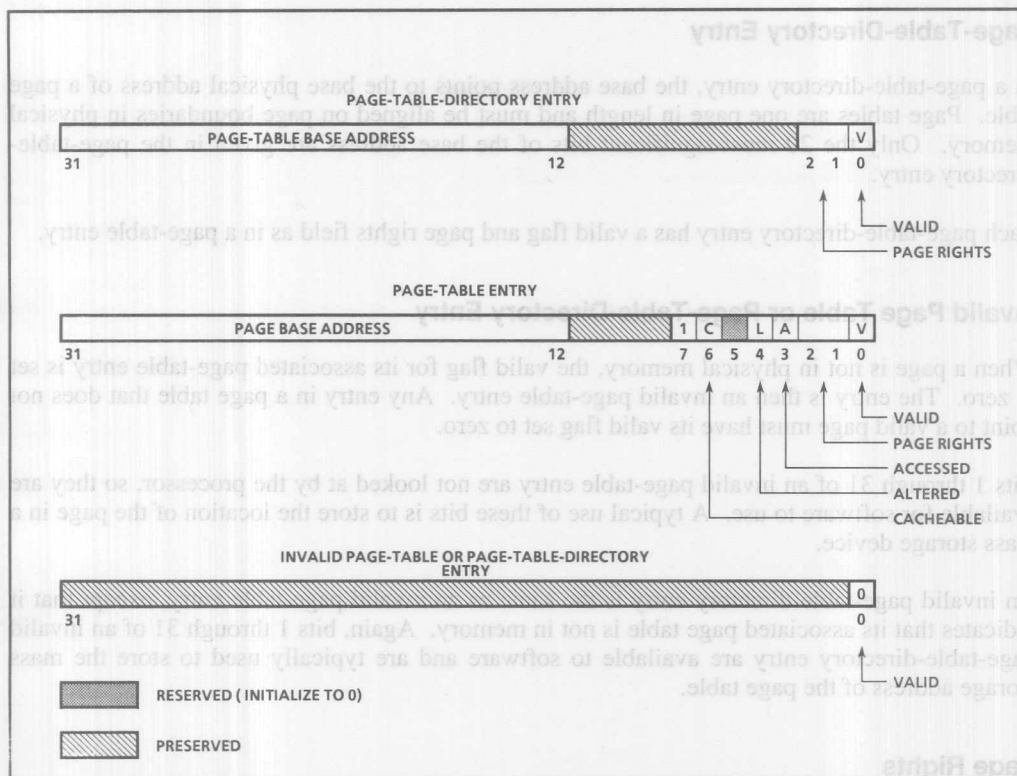


Figure 8-14: Page Table or Page-Table-Directory Entries

## Page-Table Entry

In a page-table entry, the base address points to the base physical address of a page. The page must be a full 4096 bytes in length and be aligned on a page boundary in physical memory. Only the 20 most-significant bits of the base address are given.

For paged or bipaged segments, the accessed, altered, and cacheable information is shown at the page level in the page-table entry.

Each page-table entry also has a valid flag. This flag can be either 1 or 0, depending on whether or not the page is present in physical memory. However, as described in a following section titled "Invalid Page Table Or Page-Table-Directory Entry," this flag will normally be set to 1.

The page rights field shows what operations (i.e., read or write) can be performed on the contents of the page. Page rights are discussed in a following section titled "Page Rights."

## Page-Table-Directory Entry

In a page-table-directory entry, the base address points to the base physical address of a page table. Page tables are one page in length and must be aligned on page boundaries in physical memory. Only the 20 most-significant bits of the base address are given in the page-table-directory entry.

Each page-table-directory entry has a valid flag and page rights field as in a page-table entry.

## Invalid Page Table or Page-Table-Directory Entry

When a page is not in physical memory, the valid flag for its associated page-table entry is set to zero. The entry is then an invalid page-table entry. Any entry in a page table that does not point to a valid page must have its valid flag set to zero.

Bits 1 through 31 of an invalid page-table entry are not looked at by the processor, so they are available for software to use. A typical use of these bits is to store the location of the page in a mass storage device.

An invalid page-table-directory entry is the same as an invalid page-table entry, except that it indicates that its associated page table is not in memory. Again, bits 1 through 31 of an invalid page-table-directory entry are available to software and are typically used to store the mass storage address of the page table.

## Page Rights

When operating in virtual-addressing mode, the processor allows access to information in physical memory to be restricted on a page by page basis. The page rights field in the page table and page-table-directory entries determines the access rights for a particular page or group of pages, respectively.

The processor checks these page rights each time it accesses memory.

Three levels of access rights are allowed: no access, read-only, and read-write. The page rights bits are interpreted differently depending on the execution mode (i.e., user or supervisor) that the processor is operating in, as shown in Table 8-1.

**Table 8-1: Page Access Rights Interpretation**

Rights	Execution Mode	
	User	Supervisor
00	no access	read only
01	no access	read-write
10	read only	read-write
11	read-write	read-write

When the processor accesses a page in a paged segment (e.g., a paged region), the page rights from the page's page-table entry determine the access rights for the page. When the processor accesses a page in a bipaged segment, the minimum page rights from a page's associated page-table entry and page-table-directory entry determine the access rights for the page.

For example, in a bipaged segment, if the page rights in the page-table entry are read-write, but the page rights in the page-table-directory entry are read-only, the processor will be allowed only to read the page.

The inspect access instruction (**inspace**) returns the effective page rights of the access path for a specified address. This instruction is useful in fault handling routines.

When the processor is in physical-addressing mode, virtual address translation is turned off, which disables page rights checking.

## ADDRESS TRANSLATION IN VIRTUAL MODE

This section describes how the processor uses the memory management data structures described in the previous sections to translate an SS into the location of a segment descriptor in a segment table. It also describes how the processor translates a 32-bit virtual address into a 32-bit physical address.

### SS Translation

The processor can get an SS either from a system data structure or from an instruction operand issued by a kernel routine. Once it has received an SS, the processor translates it into an offset into the segment table. This offset is to the physical address of the least significant byte of the SS's associated segment descriptor.

As is described in the following sections, the translation is slightly different depending on whether the segment table is a small or a large table. In either case, the processor has already translated the SS for the segment table to determine the base address of the segment table itself.

### Small Segment Table SS Translation

The processor uses the following procedure to locate a segment descriptor in a small segment table:

1. If the segment index in the SS is greater than  $255_{10}$ , signal a segment-length fault.
2. Locate the segment descriptor whose base address is the base address of the segment table plus 16 times the segment index.
3. If the valid flag for the descriptor is set to 0, signal the invalid segment-descriptor fault.



### Large Segment Table SS Translation

The processor uses the following procedure to locate a segment descriptor in a large segment table:

1. If the segment index is greater than 262,143<sub>10</sub>, signal the segment-length fault.
2. Get the address of the page table from the large-segment-table segment descriptor at segment index 8.
3. Locate the page-table entry, whose word offset is given by bits 14 through 23 of the SS.
4. If the valid flag in the page-table entry is 0, signal the invalid page-table entry fault.
5. Locate the segment descriptor whose base address is the base address from the page-table entry plus 16 times bits 6 through 13 of the SS.
6. If the valid flag for the descriptor is set to 0, signal the invalid segment-descriptor fault.

### Virtual-Address Translation

The term virtual address refers to an address in the address space for the currently running process (i.e., the process address space). That address is a virtual address if the address space has been mapped into physical memory using the virtual memory mapping mechanism (i.e., region segments, page tables, and pages) described earlier in this chapter.

The processor receives addresses as operands in instructions. If the processor is operating in virtual-addressing mode, it assumes that any address it receives is a virtual address. The processor then translates the address automatically into a physical address.

Figure 8-15 shows how a virtual address is broken down into a physical address depending on whether the region that contains the address is a simple region, a paged region, or a bipaged region.

In the first step of the translation process, the processor uses bits 30-31 of the virtual address to determine which region the address is in. The processor already has SS's for the four regions of the current address space, so it uses the SS for the selected region to locate the segment descriptor for that region.

If the descriptor is an invalid segment-table entry, the invalid-descriptor fault is signaled. If the descriptor is not one for a simple, paged, or bipaged region, the action is unpredictable. If the valid flag in the descriptor is 0, the invalid segment-table entry fault is signaled.

The following procedures describe the rest of the translation process, depending on the type of region being accessed.

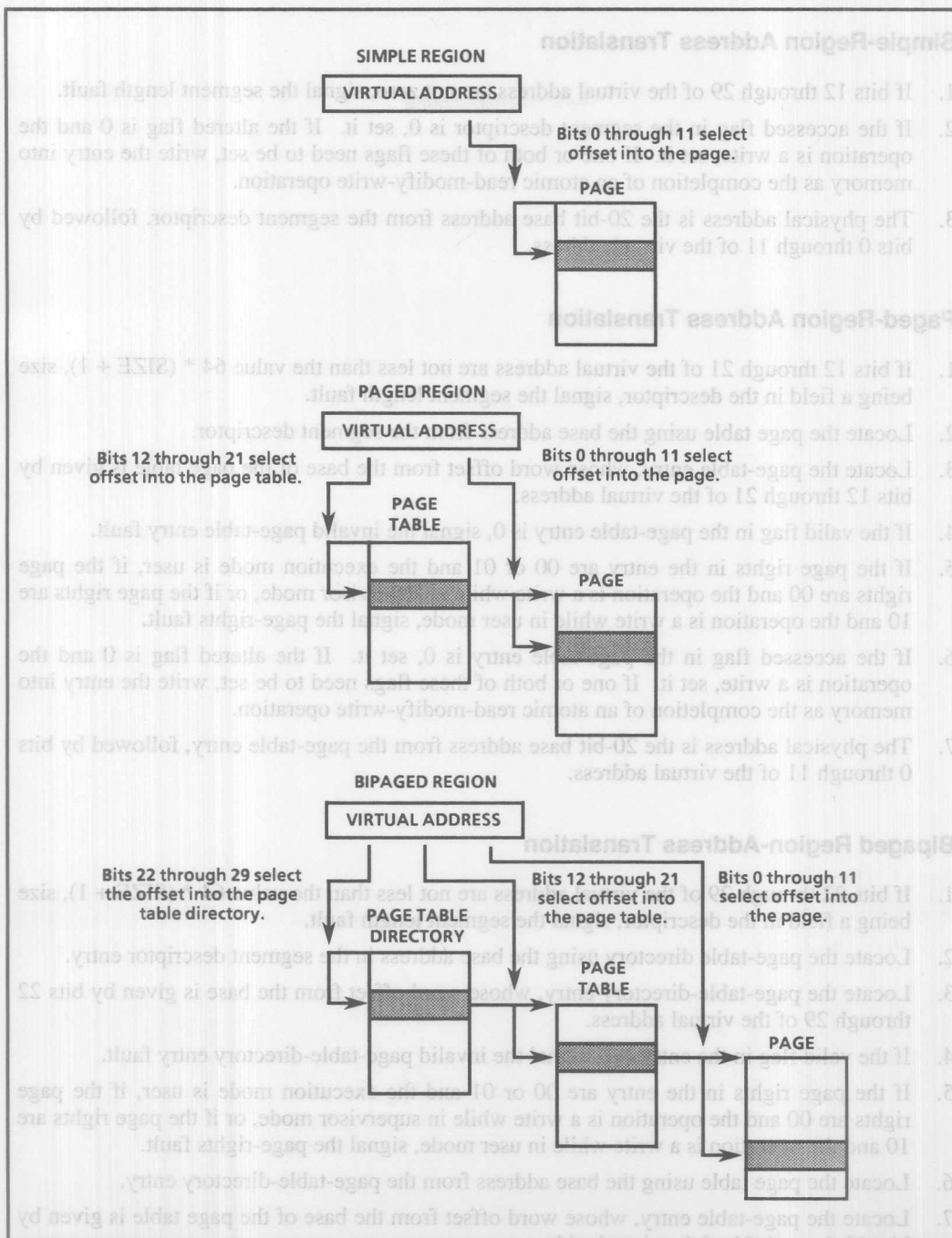


Figure 8-15: Virtual-Address Translation

### Simple-Region Address Translation

1. If bits 12 through 29 of the virtual address are not zero, signal the segment length fault.
2. If the accessed flag in the segment descriptor is 0, set it. If the altered flag is 0 and the operation is a write, set it. If one or both of these flags need to be set, write the entry into memory as the completion of an atomic read-modify-write operation.
3. The physical address is the 20-bit base address from the segment descriptor, followed by bits 0 through 11 of the virtual address.

### Paged-Region Address Translation

1. If bits 12 through 21 of the virtual address are not less than the value  $64 * (\text{SIZE} + 1)$ , size being a field in the descriptor, signal the segment length fault.
2. Locate the page table using the base address from the segment descriptor.
3. Locate the page-table entry, whose word offset from the base of the page table is given by bits 12 through 21 of the virtual address.
4. If the valid flag in the page-table entry is 0, signal the invalid page-table entry fault.
5. If the page rights in the entry are 00 or 01 and the execution mode is user, if the page rights are 00 and the operation is a write while in supervisor mode, or if the page rights are 10 and the operation is a write while in user mode, signal the page-rights fault.
6. If the accessed flag in the page-table entry is 0, set it. If the altered flag is 0 and the operation is a write, set it. If one or both of these flags need to be set, write the entry into memory as the completion of an atomic read-modify-write operation.
7. The physical address is the 20-bit base address from the page-table entry, followed by bits 0 through 11 of the virtual address.

### Bipaged Region-Address Translation

1. If bits 22 through 29 of the virtual address are not less than the value  $64 * (\text{SIZE} + 1)$ , size being a field in the descriptor, signal the segment length fault.
2. Locate the page-table directory using the base address in the segment descriptor entry.
3. Locate the page-table-directory entry, whose word offset from the base is given by bits 22 through 29 of the virtual address.
4. If the valid flag in the entry is 0, signal the invalid page-table-directory entry fault.
5. If the page rights in the entry are 00 or 01 and the execution mode is user, if the page rights are 00 and the operation is a write while in supervisor mode, or if the page rights are 10 and the operation is a write while in user mode, signal the page-rights fault.
6. Locate the page table using the base address from the page-table-directory entry.
7. Locate the page-table entry, whose word offset from the base of the page table is given by bits 12 through 21 of the virtual address.
8. If the valid flag in the page-table entry is 0, signal the invalid page-table entry fault.

9. If the page rights in the entry are 00 or 01 and the execution mode is user, if the page rights are 00 and the operation is a write while in supervisor mode, or if the page rights are 10 and the operation is a write while in user mode, signal the page-rights fault.
10. If the accessed flag in the page-table entry is 0, set it. If the altered flag is 0 and the operation is a write, set it. If one or both of these flags need to be set, write the entry into memory as the completion of an atomic read-modify-write operation.
11. The physical address is the 20-bit base address from the page-table entry, followed by bits 0 through 11 of the virtual address.

### Load Physical Address Instruction

The load physical address instruction (**ldphy**) returns a physical address for a given virtual address. This instruction allows the kernel to determine the physical address of specific data structures when only the virtual address is known.

### Spanning Page, Region, and Address-Space Boundaries

Page boundaries are completely transparent, except in cases where a memory access spans a page boundary and the pages have different rights. For example, if one page has read-write access and the adjacent page has read-only access, a write operation that spans the page boundaries will fault when it gets to the read-only page.

Region boundaries are not transparent, because each region is mapped with a different segment descriptor and page table (or set of page tables). Multiple-byte accesses that cross region boundaries can thus cause unpredictable results. This limitation can be circumvented by mapping two or more regions with the same set of page tables. This technique is described in detail later in this chapter in the section titled "Making Region Boundaries Transparent."

#### NOTE

When a multiple-byte access spans the  $2^{32}$ -byte boundary of the address space, the address wraps around to zero.

### Translation Look-Aside Buffer

To make the virtual-to-physical address translation mechanism more efficient, the processor provides a special buffer to hold address-translation information. This buffer is called the translation look-aside buffer (TLB).

When the processor receives a virtual address to be translated, it first looks in the TLB to see if it has already been translated. If it has, the processor skips the translation process and takes the physical address from the TLB.

The information stored in the TLB includes the following:

- Segment descriptors for the segment-table segment and the region-3 segment

- Segment descriptors for the current PCB segment and the region-0, -1, and -2 segments
- The page-table entry for the page that contains the bottom of the interrupt stack
- Page-table entries for pages that have been addressed at some point in the control flow of the processor

Page-table-directory entries are not stored in the TLB.

Several IACs are provided for flushing (i.e., invalidating) specified entries in the TLB to insure that it is consistent with the current state of the segment table and page tables. These IAC messages are described in Chapter 12.

## OPERATING-SYSTEM CONSIDERATIONS

The preceding discussion of the processor's virtual-memory mechanism describes the data structures required to support virtual memory and how the processor uses these structures to translate virtual addresses into physical addresses. For this mechanism to work, however, the kernel must set up and maintain these memory-management data structures.

This section suggests some ways to configure the memory-management data structures and the kernel to allow convenient management of the virtual memory system.

### Address Space Structure

Of the four regions that make up the address space, the first three regions are specific to the currently running process. The processor gets the SS's for these regions from the PCB for the current process. The fourth region is shared by all processes. The processor gets the SS for this region from the processor control block (PRCB).

#### NOTE

The PRCB is an architecture-defined data structure. It is described in Chapter 9.

Figure 8-16 shows an example of how these regions might be used to best advantage.

The address space is divided into regions primarily to improve performance in multitasking applications that require a lot of process switching. For example, if the kernel is placed in region 3, it can be shared by all processes. It can then remain in memory on a process switch, which saves page swapping time. The kernel can also be protected from the various application programs running on the system by defining the access rights for the whole of region 3 as supervisor only.

The availability of regions also facilitates the separation and protection of the major parts of an application program running in the current process. Figure 8-16 shows an example of how the code (or program text), the static data, the heap (dynamically allocated data), and the stacks (user and supervisor) might be placed in regions 0, 1, and 2.



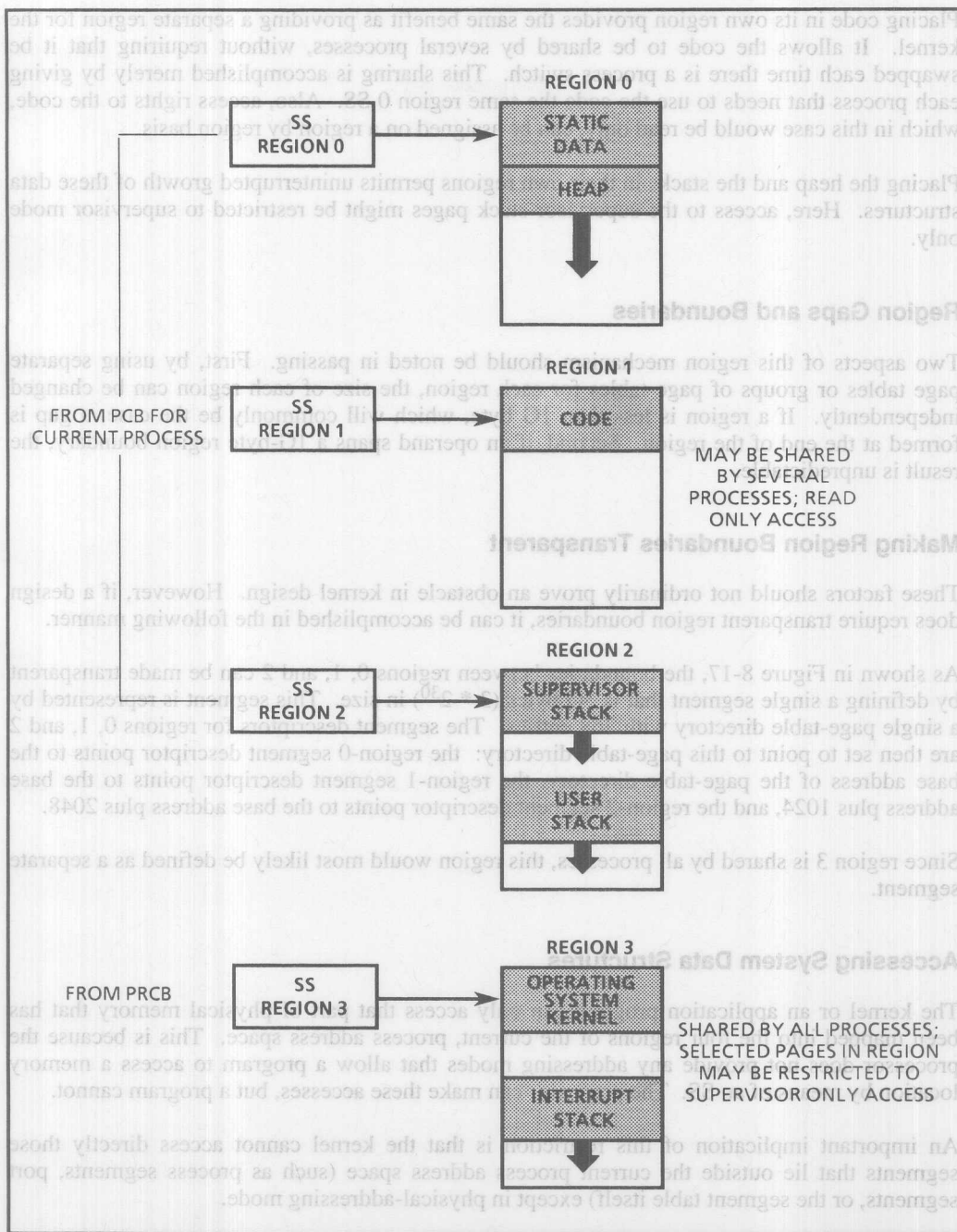


Figure 8-16: Address Space Structure



Placing code in its own region provides the same benefit as providing a separate region for the kernel. It allows the code to be shared by several processes, without requiring that it be swapped each time there is a process switch. This sharing is accomplished merely by giving each process that needs to use the code the same region 0 SS. Also, access rights to the code, which in this case would be read only, can be assigned on a region by region basis.

Placing the heap and the stacks in their own regions permits uninterrupted growth of these data structures. Here, access to the supervisor-stack pages might be restricted to supervisor mode only.

### Region Gaps and Boundaries

Two aspects of this region mechanism should be noted in passing. First, by using separate page tables or groups of page tables for each region, the size of each region can be changed independently. If a region is less than 1G byte, which will commonly be the case, a gap is formed at the end of the region. Second, if an operand spans a 1G-byte region boundary, the result is unpredictable.

### Making Region Boundaries Transparent

These factors should not ordinarily prove an obstacle in kernel design. However, if a design does require transparent region boundaries, it can be accomplished in the following manner.

As shown in Figure 8-17, the boundaries between regions 0, 1, and 2 can be made transparent by defining a single segment that is 3G bytes ( $3 * 2^{30}$ ) in size. This segment is represented by a single page-table directory with 768 entries. The segment descriptors for regions 0, 1, and 2 are then set to point to this page-table directory: the region-0 segment descriptor points to the base address of the page-table directory, the region-1 segment descriptor points to the base address plus 1024, and the region-2 segment descriptor points to the base address plus 2048.

Since region 3 is shared by all processes, this region would most likely be defined as a separate segment.

### Accessing System Data Structures

The kernel or an application program can only access that part of physical memory that has been mapped into the four regions of the current, process address space. This is because the processor does not provide any addressing modes that allow a program to access a memory location by means of an SS. The processor can make these accesses, but a program cannot.

An important implication of this restriction is that the kernel cannot access directly those segments that lie outside the current process address space (such as process segments, port segments, or the segment table itself) except in physical-addressing mode.

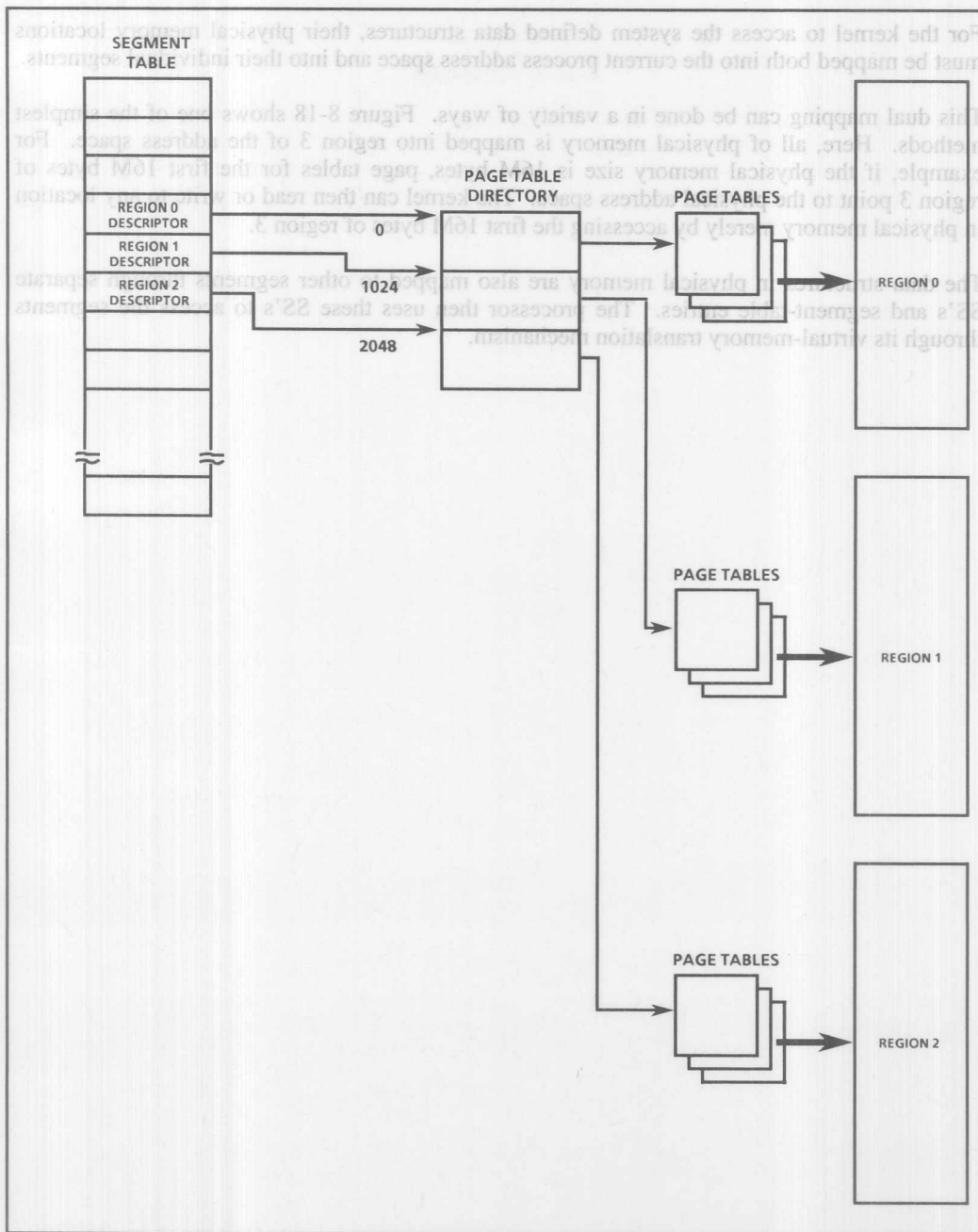


Figure 8-17: Making Region Boundaries Transparent

For the kernel to access the system defined data structures, their physical memory locations must be mapped both into the current process address space and into their individual segments.

This dual mapping can be done in a variety of ways. Figure 8-18 shows one of the simplest methods. Here, all of physical memory is mapped into region 3 of the address space. For example, if the physical memory size is 16M bytes, page tables for the first 16M bytes of region 3 point to the physical address space. The kernel can then read or write to any location in physical memory merely by accessing the first 16M bytes of region 3.

The data structures in physical memory are also mapped to other segments through separate SS's and segment-table entries. The processor then uses these SS's to access the segments through its virtual-memory translation mechanism.

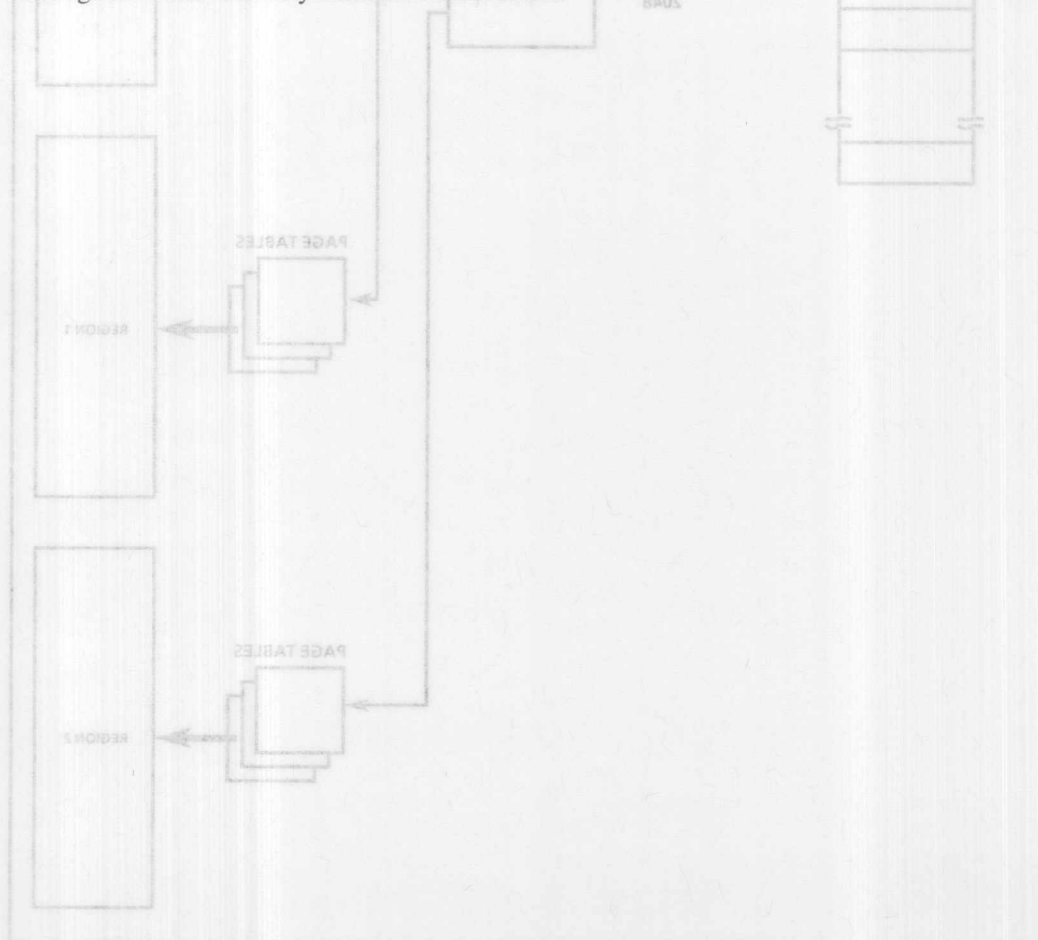


Figure 8-17: Making Region Boundaries Transparent

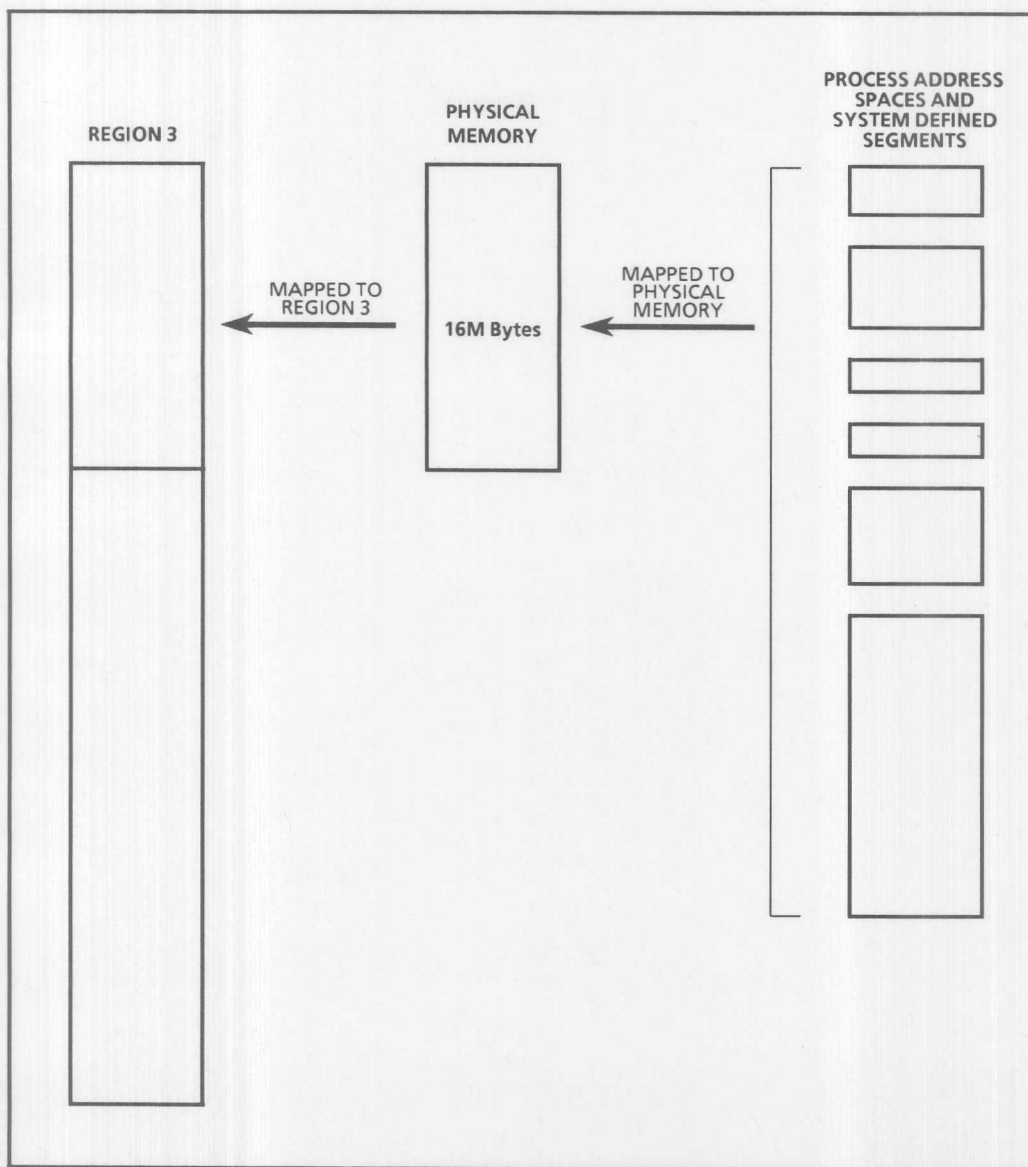


Figure 8-18: Mapping of Physical Memory to Region 3

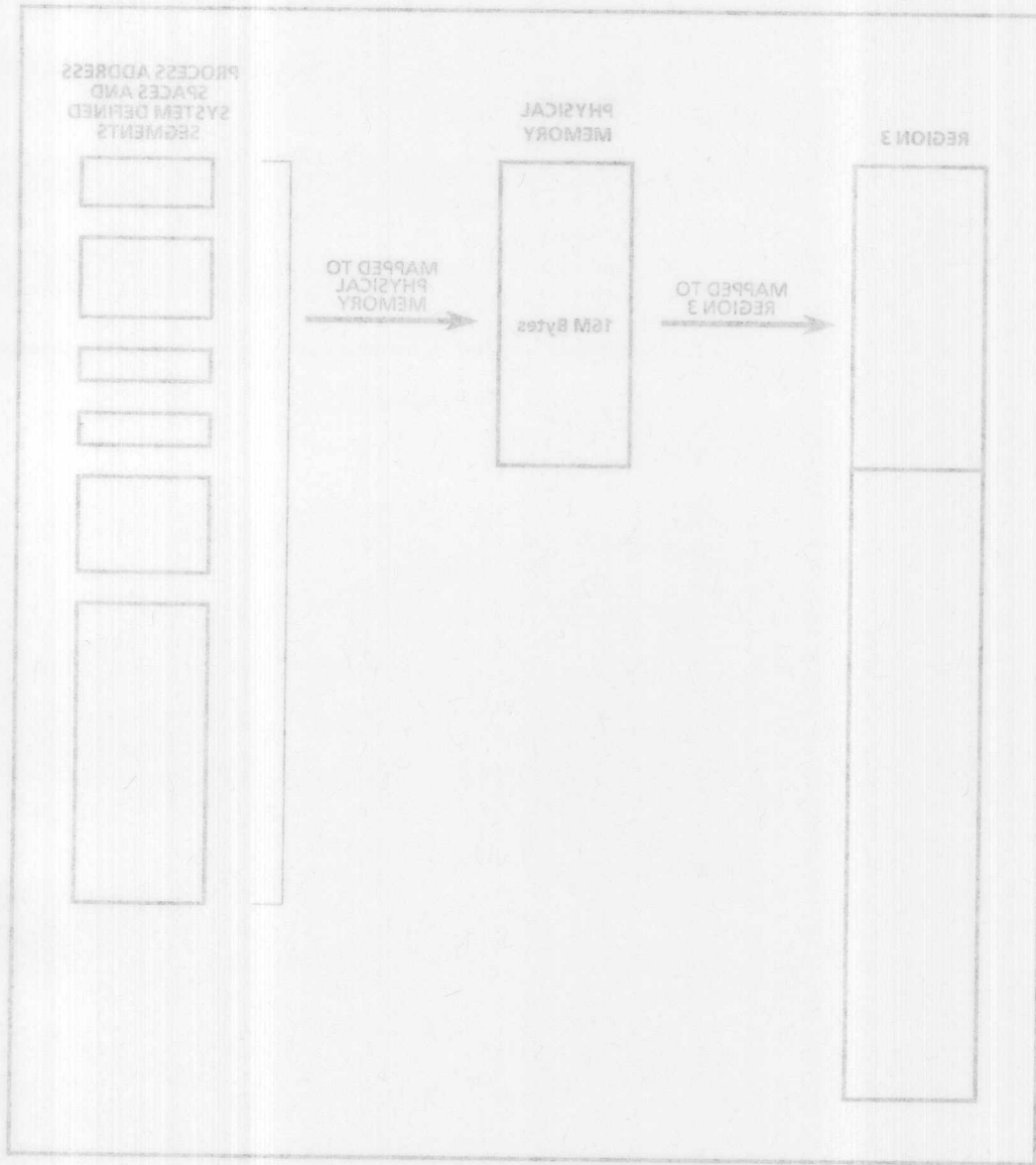


Figure 8-18: Mapping of Physical Memory to Region 3

---

# *Processor Management and Initialization*

---

**9**





## CHAPTER 9 PROCESSOR MANAGEMENT AND INITIALIZATION

This chapter describes the facilities for initializing and managing the operation of the 80960MC processor. Included is an overview of the processor-management facilities and a description of the processor-control block (PRCB). The steps required to initialize the processor are also given.

### OVERVIEW OF PROCESSOR CONFIGURATIONS

The 80960MC processor has been designed for use in a variety of system configurations. For the purpose of discussion in this manual, the possible configurations have been divided into the following three groups:

- **Single-Task System** -- Single processor performs a single task, often running from a ROM-based operating system kernel and application program.
- **Multitasking System** -- Single processor is able to perform several tasks concurrently.
- **Multiprocessing System** -- Multiple processors are able to perform several tasks, with the possibility of some tasks being processed simultaneously.

This chapter and the following chapters describe the processor and process management facilities the 80960MC processor provides. These facilities allow one or more 80960MC processors to be configured for any of the above applications. The facilities discussed are primarily software related, although some hardware considerations are also discussed.

The processor-management facilities are described in this chapter and in Chapters 10, 11, and 12. The process management facilities that support multitasking systems are described in Chapters 13 and 14. Chapter 15 describes the process and processor management facilities that support multiple-processor configurations.

### PROCESSES AND TASKS

In this manual, the terms *process* and *task* are used somewhat synonymously; however, a slight distinction between the two words should be noted. The term process refers to a unit of work that the processor is able to schedule and work on. A process is defined by information contained in a process control block (PCB).

The term task is a more general term that refers to units of work that can be scheduled at either the processor or the operating-system kernel level. For example, a multitasking system is one that performs multiple tasks. Each task may be presented to the processor in the form of a process with its own PCB. Or, each task may be scheduled and dispatched in software, with all the tasks executed in the context of a single process.

## PROCESSOR-MANAGEMENT FACILITIES

The following processor-management facilities are used to initialize, communicate with, and control the processor:

- Instruction List
- System Data Structures
- Interrupts
- IACs
- Faults
- Process Scheduling and Dispatching

These facilities allow system hardware and the operating system or kernel to initialize the processor and initiate instruction execution. They also provide software or external agents with methods of interrupting the processor to change jobs or to service external I/O devices. In more advanced systems, these facilities provide a means of synchronizing multiple tasks and multiple processors.

The following paragraphs give an overview of these processor-management facilities.

### Instruction List

At the most rudimentary level, the processor is controlled through a stream of instructions that the processor fetches from memory and executes one at a time. Once the processor is initialized, it begins executing instructions and continues until it is stopped or goes into an idle state.

### System Data Structures

The processor requires several system data structures that reside in memory. These data structures offer a means of configuring the processor to operate in a specific way. They also contain state information that the processor and kernel use to keep track of processor and process management functions.

Figure 9-1 shows the system data structures required to run a single process, using the virtual-addressing mode. In this illustration, the dashed lines indicate physical-address pointers and the solid lines indicate SS pointers.

The processor contains pointers to two of these data structures: the processor-control block (PRCB) and the segment table. The PRCB contains setup information for the processor itself and pointers to the other system data structures that the processor must access. There is one PRCB for each processor in a system.

The segment table provides address translation information for virtual-memory management, as described in Chapter 8. It should be noted that even though a segment table is not generally used when using strictly physical addressing, a rudimentary segment table is required to initialize the processor. This initialization segment table is described later in this chapter in the section titled "Processor Initialization."

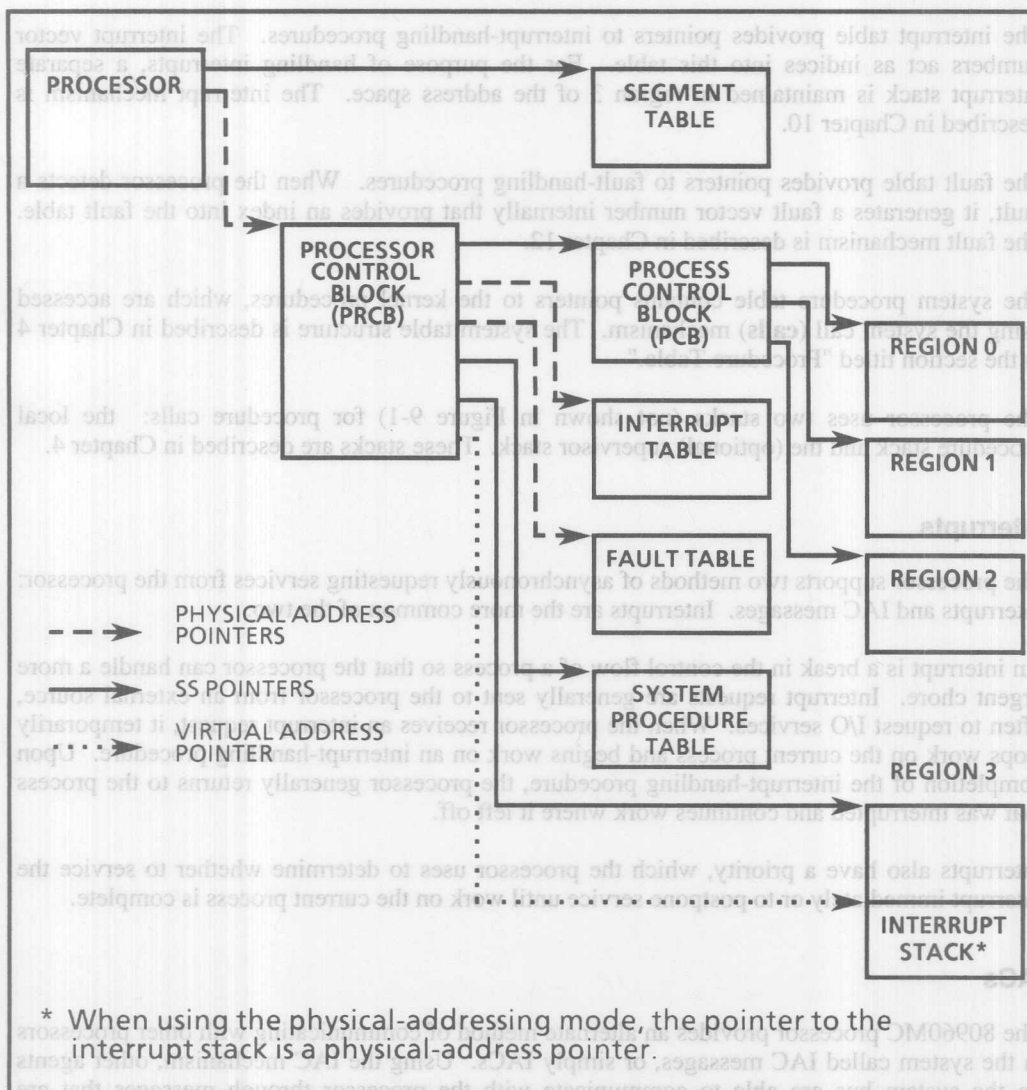


Figure 9-1: System Data Structures

Figure 9-1 shows the pointer from the processor to the segment table as an SS pointer. When initializing a processor, the first segment-table pointer that the processor receives is a physical-address pointer. (This pointer is supplied in the initial memory image.) It uses this physical address to get the SS pointer that it uses from then on. Even when using strictly physical addressing, the pointer to the segment table is always an SS pointer.

The PCB contains state information and processing requirements for the currently running process. In multiple-process systems, each process has its own PCB. A major function of the PCB is to provide pointers to regions 0, 1, and 2 of the address space for the process. (The pointer to Region 3 is given in the PRCB.) The PCB is described in detail in Chapter 13.

The interrupt table provides pointers to interrupt-handling procedures. The interrupt vector numbers act as indices into this table. For the purpose of handling interrupts, a separate interrupt stack is maintained in region 3 of the address space. The interrupt mechanism is described in Chapter 10.

The fault table provides pointers to fault-handling procedures. When the processor detects a fault, it generates a fault vector number internally that provides an index into the fault table. The fault mechanism is described in Chapter 12.

The system procedure table contains pointers to the kernel procedures, which are accessed using the system call (**calls**) mechanism. The system table structure is described in Chapter 4 in the section titled "Procedure Table."

The processor uses two stacks (not shown in Figure 9-1) for procedure calls: the local procedure stack and the (optional) supervisor stack. These stacks are described in Chapter 4.

## Interrupts

The processor supports two methods of asynchronously requesting services from the processor: interrupts and IAC messages. Interrupts are the more common of the two.

An interrupt is a break in the control flow of a process so that the processor can handle a more urgent chore. Interrupt requests are generally sent to the processor from an external source, often to request I/O services. When the processor receives an interrupt request, it temporarily stops work on the current process and begins work on an interrupt-handling procedure. Upon completion of the interrupt-handling procedure, the processor generally returns to the process that was interrupted and continues work where it left off.

Interrupts also have a priority, which the processor uses to determine whether to service the interrupt immediately or to postpone service until work on the current process is complete.

## IACs

The 80960MC processor provides an alternate method of communicating with other processors in the system called IAC messages, or simply IACs. Using the IAC mechanism, other agents on the system bus are able to communicate with the processor through messages that are exchanged in a reserved section of memory.

Like interrupts, IACs are used to request that the processor stop work on the current process and begin work on another chore. However, where an interrupt generally causes a temporary break in the execution of a process, an IAC often causes a permanent change in the control flow of the processor. An important application of IACs is to coordinate the activities of multiple processors.

The IAC mechanism is described in Chapter 11.

## Faults

While executing instructions, the processor is able to recognize certain conditions that could cause it to return an inappropriate result or that could cause it to go down a wrong and possibly disastrous path. One example of such a condition is a divisor operand of zero in a divide operation. Another example is an attempt to access a memory location in a page that is not in physical memory. These conditions are called faults.

The processor handles faults almost the same way that it handles interrupts. When the processor detects a fault, it automatically stops its current processing activity and begins work on a fault-handling procedure.

## Process Scheduling and Dispatching

The processor also provides some advanced process-management facilities that are able to signal the processor internally to suspend the process it is currently working on and begin work on another process. These features, which are useful in the scheduling and dispatching of processes, are described in Chapter 14.

## PROCESSOR-CONTROL BLOCK

The processor is controlled through the PRCB, which contains information related to the processor's operation. The PRCB is 176 bytes in length and is contained in physical memory, not in a segment. Each CPU processor in a 80960MC-based system has its own PRCB. The processor locates and reads its PRCB at initialization by means of a physical-address pointer to the first byte of the block.

The processor caches parts of the PRCB on chip and updates these cached fields internally. After the processor has initially cached these fields, it does not check or update the original PRCB in memory. IACs are provided that allow those parts of the PRCB that the processor has copied into internal storage to be changed. These IACs are discussed later in this chapter in the section titled "Changing the PRCB." This section also lists the fields of the PRCB that are cached on the chip.

Figure 9-2 shows the structure of the PRCB and Figure 9-3 shows the structure of the *processor-controls* word in the PRCB. The following paragraphs describe the use of each of the fields in the PRCB.

### Processor-Controls Word

The *processor-controls* word contains several miscellaneous pieces of information to control processor operation. The function of the various fields in this word are as follows.

The *multiprocessor-preempt* flag, when set, enables a high-level process preemption function that allows multiple processors to handle preempting processes. This function is only useful in multiple-processor systems and should be set to 0 for single-processor systems. Refer to the section titled "Process Preemption" in Chapter 14 for more information on this function.



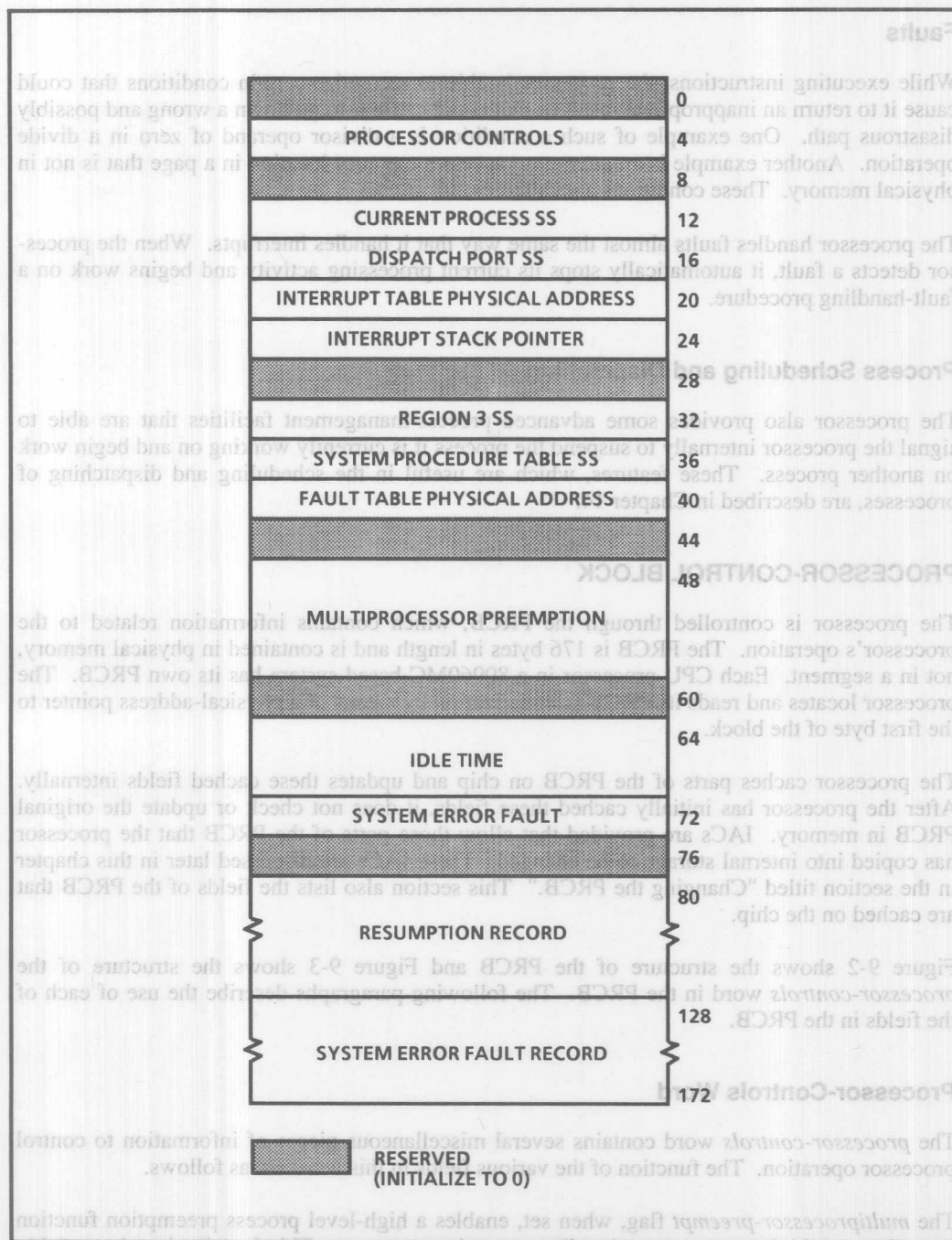


Figure 9-2: Processor-Control Block (PRCB)

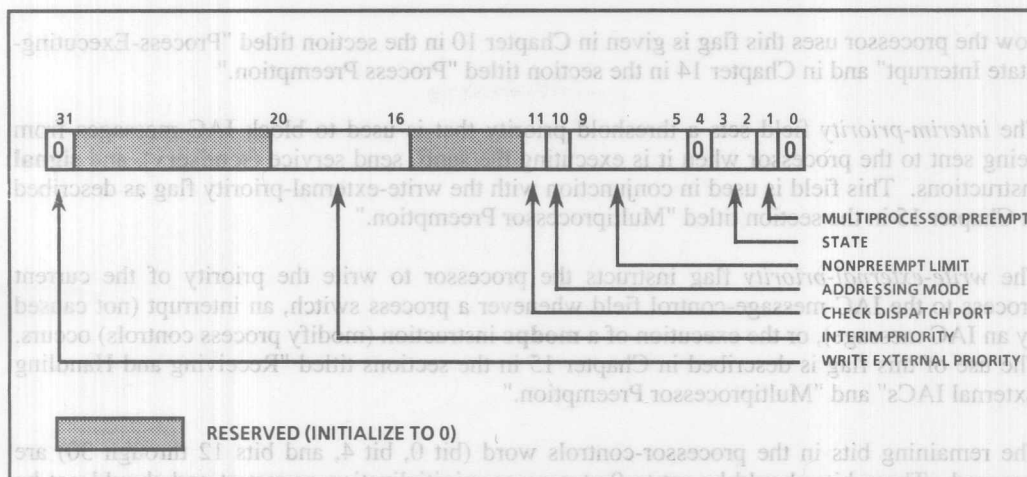


Figure 9-3: Processor-Controls Word

The *state* field determines the state of the processor when it is initialized or restarted. The encoding of this field is shown in Table 9-1.

Table 9-1: Encoding of the State Field

State Field	State
00	Stopped
01	Reserved
10	Idle
11	Process executing

The section later in this chapter titled "Processor and Process States" describes the activities of the processor while it is in these different states.

The *nonpreempt-limit* field sets a threshold priority that the processor uses in determining whether or not to allow one process to preempt another. If the priority of the preempting process is at or below that of the current process or the nonpreempt limit, the processor will not preempt the current process. This field is used during process preemption and on returns from interrupts. Further discussion of this limit is given in Chapter 10 in the section titled "Process-Executing-State Interrupts" and in Chapter 15 in the section titled "Multiprocessor Preemption."

The *addressing-mode* flag determines the address-translation mode of the processor: physical addressing (0), or virtual addressing (1). The section later in this chapter titled "Address-Translation Modes" discusses these modes.

The *check-dispatch-port* flag instructs the processor to check the dispatch port for processes of higher priority than the current process, during returns from interrupts. Only the processor uses this flag. Software should set it to 0 at processor initialization or restart. A discussion of

how the processor uses this flag is given in Chapter 10 in the section titled "Process-Executing-State Interrupt" and in Chapter 14 in the section titled "Process Preemption."

The *interim-priority* field sets a threshold priority that is used to block IAC messages from being sent to the processor when it is executing the **send**, send service (**sendserv**), and **signal** instructions. This field is used in conjunction with the write-external-priority flag as described in Chapter 15 in the section titled "Multiprocessor Preemption."

The *write-external-priority* flag instructs the processor to write the priority of the current process to the IAC-message-control field whenever a process switch, an interrupt (not caused by an IAC message), or the execution of a **modpc** instruction (modify process controls) occurs. The use of this flag is described in Chapter 15 in the sections titled "Receiving and Handling External IACs" and "Multiprocessor Preemption."

The remaining bits in the processor-controls word (bit 0, bit 4, and bits 12 through 30) are reserved. These bits should be set to 0 at processor initialization or restart and should not be altered after that.

Figure 9-3: Processor-Controls Word

## System-Data-Structure Pointers

As is shown in Figure 9-1, the PRCB contains pointers to several system data structures, which are summarized in the following paragraphs.

The *current-process-SS* field points to the PCB for the process that is currently bound to the processor. (The mechanism for binding a process to the processor is described in Chapters 13 and 14.)

If the processor is restarted in the process-executing state, the processor binds itself to the process specified in the current-process-SS field. For single process systems this is the only process bind action that is carried out.

For systems that execute multiple processes, the current-process-SS field is updated each time a new process is dispatched and bound to the processor.

When the processor is not in the process-executing state, this field is not used. Also, this field is not cached on the processor chip.

The *dispatch port SS* field points to the dispatch port that the processor goes to for new processes during a dispatching operation. This field is only used for multiple process systems that use the processor's high-level process management functions.

The *interrupt table physical address* points to the first byte of the interrupt table.

The *interrupt-stack pointer* points to the top (first available byte) of the interrupt stack. In the virtual-addressing mode, the processor interprets the interrupt-stack pointer as a virtual address in the current-process address space. (When using the virtual-addressing mode, the interrupt stack should be placed in region 3 of the process address space.) When using the physical-addressing mode, the interrupt-stack pointer is interpreted as a physical address.

The *region 3 SS* points to the region segment that contains region 3.

The *system procedure-table SS* points to the system procedure table.

The *fault-table physical address* points to the first byte of the fault table.

### Miscellaneous PRCB Fields

The following fields in the PRCB provide miscellaneous processor-control functions.

The *idle time* field contains a long ordinal that gives the time that the processor has spent in the idle or idle-interrupted state. Idle timing is discussed later in this chapter in the section titled "Idle Timing."

When a system-error fault occurs, the type and subtype of the fault are stored in bits 16 through 23 and bits 0 through 7 of the *system error fault* field, respectively. The fault record is stored in the *system-error fault record* field. System-error faults are described in Chapter 12.

The *resumption record* field contains the intermediate state of an instruction that has been interrupted. This information is generally stored in the PCB for the interrupted process. However, when the processor is interrupted while in the idle-interrupted state, the resumption information is stored in the PRCB. This field should be set to all zeros at initialization or restart of the processor and not accessed by software thereafter.

The processor uses *multiprocessor preemption* field while handling preempting processes in multiprocessor applications. The use of this field is described in Chapter 15 in the section titled "Preemption Control."

The remaining fields in the PRCB (bytes 8 through 11, bytes 28 through 31, bytes 44 through 47, bytes 60 through 63, and bytes 76 through 79) are reserved. They should be set to all zeros at initialization or restart and not accessed by software thereafter.

### Changing the PRCB

At initialization, on a restart processor IAC, or on a warmstart processor IAC, the processor reads the following fields from the PRCB in memory and caches them:

- Processor controls
- Dispatch port SS
- Interrupt table physical address
- Interrupt stack pointer
- Region 3 SS
- System procedure table SS
- Fault table physical address
- Idle time

In general, to change any of the PRCB fields that have been cached on the processor chip, the kernel must first modify the PRCB in memory, then restart the processor using the restart processor IAC. The processor then rereads the PRCB and reloads the cached fields in its internal cache.

The store processor IAC may also be useful here. This IAC causes any of the cached parts of the PRCB that have been changed since initialization or the last restart to be written to the PRCB in memory. Software is thus able to examine the current state of the PRCB, modify it, then restart the processor.

The modify-processor-controls IAC allows any of the fields in the processor-controls word, except the state field, to be changed without restarting the processor. If this IAC is used to change the state field, the processor must be restarted for the change in state to become effective.

## PRIORITIES

The processor provides a priority mechanism for determining the order in which processes and interrupts are worked on. Priorities range from 0 to 31, with 31 being the highest priority. Each process and interrupt vector is assigned a priority.

In multitasking systems, process priorities are used to determine which processes are worked on first. Process priorities also allow a process of higher priority to preempt the current process if the current process has a lower priority. The term preempt means that the current process is suspended and the preempting process is bound to the processor.

Interrupt priorities serve two functions. First, they determine if the processor will service an interrupt immediately or delay servicing it with respect to the priority of the current process. Second, they determine which interrupt of several interrupts is serviced first.

The processor always handles an IAC as soon as it is received (i.e., IACs are assumed to have a priority of 31). However, in certain system designs IACs can be prioritized. Here, external hardware is required to compare the priority of the IAC with that of the current process, then determine whether to send the IAC message to the processor immediately or reject it. The M82965 is designed to perform this operation.

## PROCESSOR AND PROCESS STATES

The processor has three different operating states: process executing, idle, and stopped. In addition, a process can be in either of two states: executing and interrupted. When the processor and process states are combined, five states are possible for the processor and its current process: process executing, process interrupted, idle, idle interrupted, and stopped. The processor is placed in one of three states (process executing, idle, or stopped) at initialization or restart. After that, the processor and software control the state of the processor and process.

The processor can switch between the process-executing, process-interrupted, idle, and idle-interrupted states. However, the processor never switches from the process-executing or idle states to the stopped state, unless a system-error fault occurs.



Software can change the state of the processor in either of two ways: (1) restart the processor in the desired state, or (2) issue a stop processor IAC message.

The following paragraphs describe the five combined processor and process states.

### Process-Executing and Process-Interrupted State

In the process-executing state, the processor is executing the process specified in the current process SS field of the PRCB.

If the processor is interrupted while in the process-executing state, it saves the state of the current process, switches to the process-interrupted state, and services the interrupt. Upon returning from the interrupt handler, the processor resumes work on the current process.

### Stopped State

In the stopped state the processor ceases all activity. The only task it can perform while in this state is to service an IAC. If the IAC handling action does not result in a change in the processor's state, the processor switches back to the stopped state when it finishes the IAC handling action.

The only way to get the processor out of the stopped state is to restart the processor in a different state.

### Idle and Idle-Interrupted States

The idle and idle-interrupted states are used primarily with the processor's high-level process-management functions. Here, the processor switches to the idle state when it examines the dispatch port and finds no processes available for processing. The idle state eliminates the need for the kernel to provide a special process for the processor to run when no other processes are scheduled.

Note that even if a process is available at the dispatch port, the processor is considered to be in the idle state while it is "between" processes (i.e., after suspending the current process and before dispatching another process).

The processor may be interrupted while in the idle state. While servicing the interrupt, the processor switches to the idle-interrupted state. Upon completion of the interrupt routine, the processor returns to the idle state and begins polling the dispatch port again for a process to run.

While in the idle state, the processor polls the dispatch port once every tick (i.e., once every 256 clock cycles). When running at a 16-MHz clock rate, this polling rate equates to once every 8 microseconds. (Refer to the section later in this chapter titled "Processor Timing" for more information on ticks.)



The other use of the idle state is at initialization. During the first stage of initialization, the processor is placed in the idle state. From there, the processor goes into the idle-interrupted state to execute initialization code.

If a system does not have a dispatch port, the processor will never go into the idle state except at initialization. If the processor is restarted in the idle state when there is no dispatch port, the behavior of the processor is unpredictable.

## ADDRESS-TRANSLATION MODES

As was discussed in Chapter 8, the processor can operate in either of two address-translation modes: physical-addressing mode and virtual-addressing mode. The addressing-mode flag in the processor controls determines which address-translation mode the processor is using.

These modes only apply to the translation of addresses in the address space for the current process. In the physical-addressing mode, all addresses are assumed to be physical addresses and are sent out on the bus unchanged. In the virtual-addressing mode, addresses are assumed to be virtual addresses. The processor memory-management unit (MMU) then translates these addresses into physical addresses before they are sent out on the bus.

Regardless of the mode, SS's are treated the same. When the processor receives an SS, it locates the selected segment in memory and uses an internally generated or explicit offset to access the selected byte in the segment. Thus, even if the processor is operating in physical-addressing mode, it still uses the SS's in the PRCB to locate system data structures. Likewise, privileged supervisor-mode instructions that use SS's as operands are treated the same way in both address-translation modes.

## Changing the Address-Translation Mode

Generally, the kernel will run the processor in one address-translation mode or the other. If strictly physical addressing of memory is used, the processor will be run in physical-addressing mode, and if a virtual-memory system is supported, the processor will run in virtual-addressing mode.

It is possible to design a system in which the address-translation mode is changed on occasion. In such instances, the change of mode can be accomplished in either of two ways.

The safest way is to establish an up-to-date image of the PRCB in memory, perhaps by using the store processor IAC. The addressing-mode flag is then changed and the processor is restarted.

The other way is to use the modify-processor-controls IAC. When this IAC is used to change the address-mode flag, the processor reads the new value and changes its mode accordingly.

Changing the address-translation mode in this manner can have serious consequences. For example, clearing the flag causes the IP for the next instruction to be interpreted as a physical address, which might take the processor down an unpredictable path. Also, the system may be maintaining a memory cache for the processor. Changing the address-translation mode would cause the cached addresses to be interpreted differently.

If the address-translation mode is to be changed in this latter manner, the safest way to do so is to map the addresses of at least the code and the stacks into the same locations in both the virtual and physical address spaces. It will be necessary to purge the instruction cache of the processor (using the purge instruction cache IAC).

## PROCESSOR TIMING

The processor provides several counting functions such as process execution timing and idle timing. Counting for these functions is in terms of ticks.

### Duration of a Tick

For the 80960MC processor, a tick is defined as 256 external clock periods (128 internal clock periods). For a 16-MHz processor clock rate (32-MHz external clock), a tick is then 8 microseconds. For a 20-MHz processor clock rate, a tick is 6.4 microseconds.

### Idle Timing

The idle time field of the PRCB is used to count the amount of time that the processor is in the idle state. When the processor goes into the idle state it begins incrementing the count in the idle time field one count for each tick. When the processor switches to another processor state, idle-time counting is stopped.

The idle time field, like others in the PRCB, may be cached in the processor. If this is the case, the value must be periodically written out to the PRCB in memory so software can read it. The interval that the processor uses to update the count is once every 32 ticks.

The processor writes the idle time field periodically, but it only reads this field at initialization. As a result, if software changes the idle time field after initialization, the count will be inconsistent.

### NOTE

If the interrupt handler sets the timing flag in the process controls word, the processor will begin counting idle time for the interrupted state. This practice is not advisable because it can cause unpredictable events, most notably an unwanted time-slice fault.

## INSTRUCTION SUSPENSION

When a process is suspended or interrupted while the processor is in the midst of executing an instruction, the processor does one of three things before it suspends the process or services the interrupt:

1. It completes the instruction.
2. It terminates the instruction and sets the process state so that it is as if execution of that instruction had not yet begun.

3. It suspends the instruction and saves the necessary resumption information so that execution of the instruction can be continued when the processor begins work on the process again. This course of action is generally reserved for instructions that have a long execution time and that alter the internal and external process state as they execute.

Which of these steps the processor takes depends on the instruction being executed. However, whichever step it takes is transparent to the software. The processor automatically saves the necessary state information so that work on the process can be resumed with no loss of information.

Refer to the section in Chapter 13 titled "Resumption Record" for more information on how resumption information for a suspended instruction is saved when a process is suspended. Refer to the section in Chapter 10 titled "Interrupt-Handling Action" for more information on how resumption information is saved when an interrupt is serviced.

## SOFTWARE REQUIREMENTS FOR PROCESSOR MANAGEMENT

The processor-management facilities described earlier in this chapter allow the processor to be configured and operated in a variety of ways. This section lists the data structures that the kernel must supply to operate the processor in a single-task configuration. (Chapter 14 lists the required data structures for a multitasking system that uses the processor's high-level process management facilities and Chapter 15 lists the requirements for a multiprocessing system.)

When using the processor in a single-task system, the kernel must provide the following items:

- Initial Memory Image
- Set of System Data Structures
- Address Space
- Stacks
- Code

The initial memory image comprises the minimum data structures that the processor needs to initialize the system. It is described later in this chapter in the section titled "Initial Memory Image."

As part of the initialization procedure, a more complete set of system data structures are established in memory. If the virtual-addressing mode of the processor is to be used, all of the data structures shown in Figure 9-1 must be set up. These data structures include a PRCB, segment table, PCB, interrupt table, interrupt stack, fault table, and the four address-space regions for the current process. If the user-supervisor protection mechanism is not going to be used, a system procedure table is not required.

### NOTE

When using the virtual-addressing mode, the kernel code and the interrupt stack would typically be located in region 3 of the process address space. However, in a single-process system, these items can be located anywhere since only one address space is used.

Structures listed above must be set up except the four address space regions and the system procedure table. The system procedure table is not required; however, it can be set up and used in a physical-addressing environment.

Note that when operating in physical-addressing mode, a segment table is still required. This segment table is part of the initial system image and is generally not used after the first stage of initialization. The required entries for this initialization segment table are given in the section later in this chapter titled "Initialization Segment Table."

Figure 9-4 shows the fields in the PRCB and the processor-controls word that are used in a single-task configuration, using the virtual-addressing mode. When using strictly physical addressing, the system procedure table SS is not required. (Chapter 10 describes the required fields for the interrupt table and interrupt stack; Chapter 12 describes the fault table; and Chapter 13 describes the PCB.)

Two stacks are required: an interrupt stack and a local (or user) procedure stack. The initial stack pointer for the interrupt stack is given in the PRCB; the initial stack pointer for the local-procedure stack is given in the local registers and is established by initialization code. If the user-supervisor protection mechanism is to be used, a supervisor stack must also be provided. The initial stack pointer for this stack is given in the system-procedure table. The supervisor stack can be placed anywhere in the address space.

Finally, three levels of code are required: initialization code, kernel code, and user (or applications) code. The initialization code is part of the initial memory image. The starting IP for the initialization code is also provided in the initial memory image. This IP will be interpreted as a physical address or a virtual address depending on the setting of the addressing-mode flag in the initial processor-controls word.

When using the virtual-addressing mode, the kernel code and user code are located in the current process-address space; when using the physical-addressing mode, this code is located in the physical address space.

The starting IP for the kernel code or the user code, whichever is run first, is provided in the RIP word in the first frame of the kernel or user stack. One of the jobs of the initialization code is thus to establish a stack in memory for the kernel or user code to use. The FP for this stack is stored in global register field g15 of the PCB.

## PROCESSOR INITIALIZATION

This section describes how to initialize the 80960MC processor. It defines the mechanism that the processor uses to establish its initial state and begin instruction execution. It also describes some general guidelines for writing code to complete the initialization of the processor for specific applications.

This initialization procedure can be used in both single-processor and multiprocessing systems. In a multiprocessing system, one processor generally performs the first stage of initialization in which an initial memory image is established and instruction execution begins.

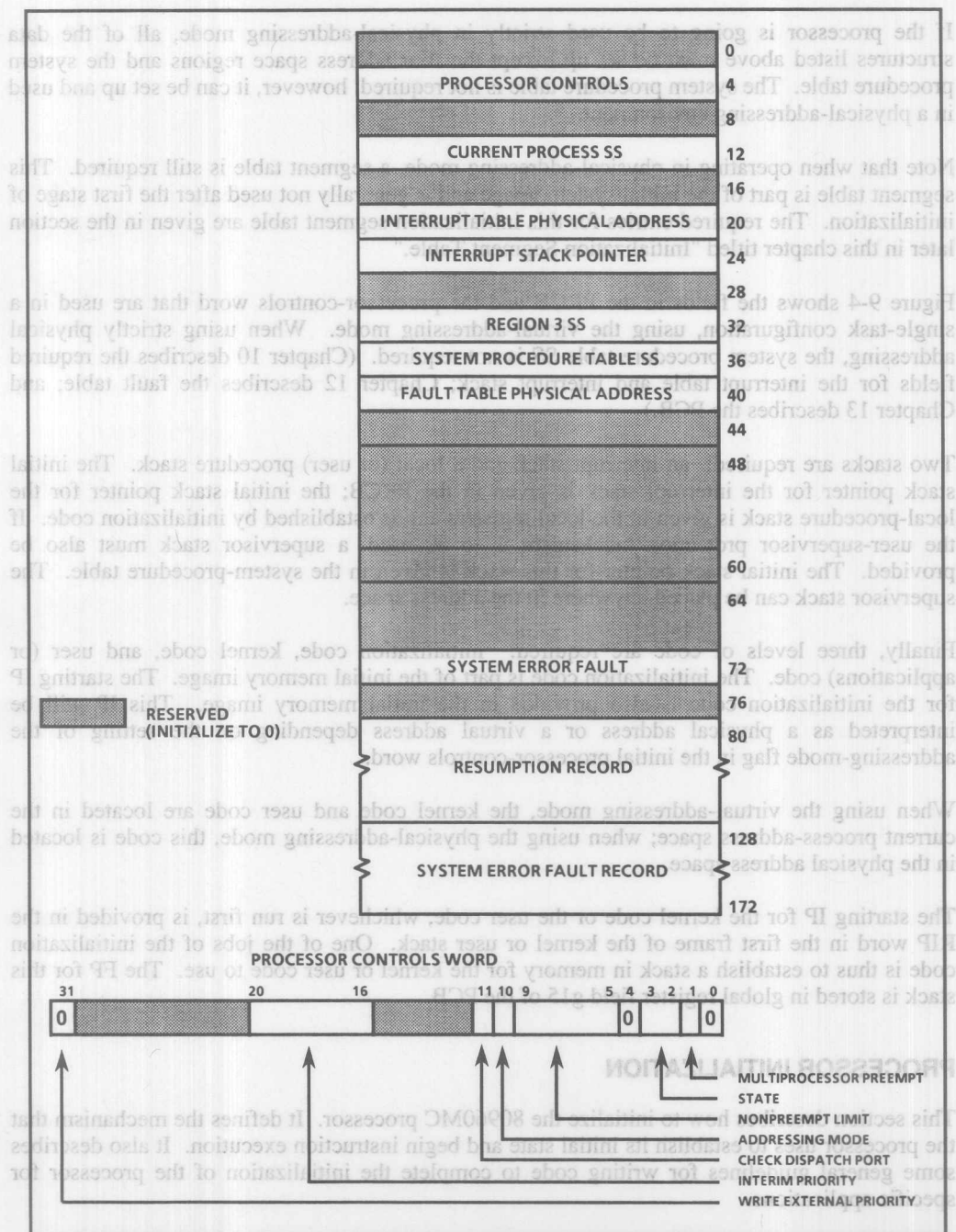


Figure 9-4: Required Fields in PRCB for Single-Task Configuration



In the next stage of initialization, the initializing processor copies additional system data structures into memory to build the memory image up to a more useful level. At this point the processor is generally restarted with this expanded memory image.

Finally, if there are additional processors in the system, the initializing processor initializes these processors by restarting them one at a time.

### Initial Memory Image

Figure 9-5 shows the minimum requirements for the memory image at initialization. This image will generally be held in ROM.

### Check-Sum Words

The first eight words (called the check-sum words) must be in physical memory locations  $00000000_{16}$  to  $0000001F_{16}$ . The first of these words is a physical-address pointer to the base of the initialization segment table. The second word is a physical-address pointer to the base of the initialization PRCB.

The fourth word is the instruction pointer to the first instruction of the initialization code. This address can be either a physical address or a virtual address, depending on the address-translation mode specified in the processor-controls word of the initialization PRCB.

The remaining words (word 3 and words 5 through 8) are check words. During the first stage of initialization of the processor, these words are added to the pointers for the initialization segment table, PRCB, and initialization code to determine a check sum. The check words must be chosen such that when the check sum is computed (as shown in initialization algorithm in Figure 9-6), the result is equal to 0.

### Initialization Segment Table

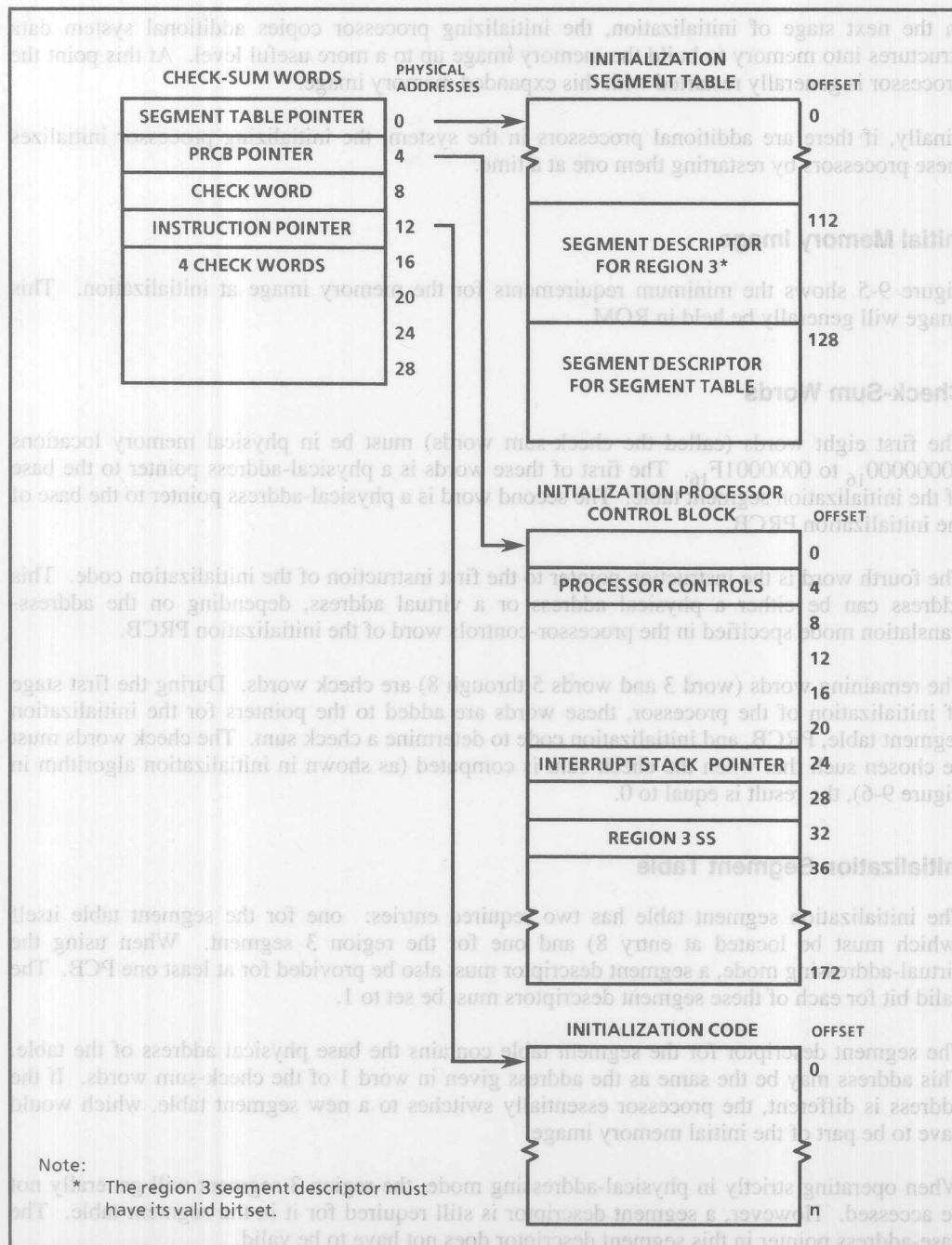
The initialization segment table has two required entries: one for the segment table itself (which must be located at entry 8) and one for the region 3 segment. When using the virtual-addressing mode, a segment descriptor must also be provided for at least one PCB. The valid bit for each of these segment descriptors must be set to 1.

The segment descriptor for the segment table contains the base physical address of the table. This address may be the same as the address given in word 1 of the check-sum words. If the address is different, the processor essentially switches to a new segment table, which would have to be part of the initial memory image.

When operating strictly in physical-addressing mode, the region 3 segment will generally not be accessed. However, a segment descriptor is still required for it in the segment table. The base-address pointer in this segment descriptor does not have to be valid.

Even though segment tables have a minimum size of 4096 bytes, only the three entries described above must be mapped into the initial memory image. Additional segment descriptors may be defined, depending on the needs of the initialization code.





**Figure 9-5: Initial Memory Image**

## Initialization PRCB

The initialization PRCB must have at least three entries: the processor-controls word, the interrupt-stack pointer, and the region-3 SS. The state field in the processor-controls word should be set to  $10_2$  (idle or idle-interrupts state).

The interrupt-stack pointer can be a virtual address in region 3 or a physical address depending on the initial address-translation mode.

Although the region-3 SS must be specified, its associated segment descriptor in the segment table does not have to point to an actual region segment in memory, unless the initialization code and interrupt stack are to be contained in this region. Note that as described in the previous section, the valid bit for the region 3 segment descriptor does have to be set.

Additional fields may be included in the PRCB, again depending on the needs of the initialization code. For example, if faults can occur during the second stage of initialization, the fault-table physical address should be valid. Likewise, if interrupts can occur, the interrupt-table physical address should be valid.

## Initialization Code

The initialization instruction list can be mapped directly to physical memory or through region 3, depending on the initial address-translation mode.

## Building a Memory Image

The initial memory image shown in Figure 9-5 contains the minimum data structures required for the processor to initialize itself and begin executing code. All of the required initialization data structures are generally stored in ROM.

To build a useful system, additional data structures are required, such as an interrupt table, a fault table, a system procedure table, a set of kernel procedures, a set of stacks, and a heap. Some of these data structures can be located in ROM along with the initial memory image; however, others must be in RAM because they must be writable.

Table 9-2 lists the various system data structures and shows which can be in ROM and which must be in RAM.

The following paragraphs give the system limitations if a data structure is included in ROM.

The segment table may be contained in ROM, providing it is not going to be changed. Otherwise, an expended segment table should be copied to RAM as part of the second stage of initialization. Also, any referenced and modified bits should already be set.

Table 9-2: ROM and RAM Resident Data Structures

Data Structure	May Be in ROM	May Be in ROM with Limitations	Must Be in RAM
Initial memory image	X		
PRCB		X	
PCB		X	
Segment table		X	
Page tables		X	
Stack and heap			X
Interrupt table			X
Fault table	X		
Kernel Procedures	X		

Part of the second stage of initialization should be to copy a new PRCB into RAM. This PRCB along with the new segment table will then be used after the processor is restarted.

The PRCB may remain in ROM; however, if it does, the following restrictions will apply:

1. Multiple processes cannot be executed. To execute multiple processes, the processor must be able to write the SS for the current process in the PRCB.
2. System-error faults will not be recoverable. On a system-error fault, the processor writes the fault record into the PRCB. If the PRCB is in ROM, this information is lost. One way around this limitation is to position the PRCB over a ROM/RAM boundary such that the fault record fields fall in RAM.
3. The processor will not be able to handle interrupts properly that occur during the execution of an instruction with long execution times. This is because a resumption record cannot be stored in the PRCB.

The PCB should be in RAM. However, if it is left in ROM, the following restrictions apply:

1. The processor will only be able to run a single process, and this process must not time out.
2. Interrupts that create a resumption record will not work properly because the record cannot be stored in the PCB.
3. The initial state of the global registers is fixed, so the stack pointer cannot be changed before the process is run.
4. The timer will not be usable since the processor periodically writes out the current value of the timer to the PCB.

Page tables are generally used to support systems that allow dynamic memory allocation, so they will generally need to be in RAM. If they are contained in ROM, paging of the address space will not be allowed, since the processor will not be able to access the valid, altered, and accessed bits.

An alternative would be to have the page tables for fixed data structures in ROM and those for address spaces or data structures that will change in RAM.

The stack, heap, and interrupt table must all be in RAM for the processor to operate properly. The reason the interrupt table must be in RAM is that it contains the interrupt pending fields, which the processor must be able to write to.

The fault table can be in ROM, providing it will never be necessary to relocate the fault handler routines.

The kernel procedures can be in either ROM or RAM or both, depending on the design of the kernel.

## TYPICAL INITIALIZATION SCENARIO

Initialization of the 80960MC processor typically is handled in two stages. In the first stage of initialization, the processor performs a self test and reads pointers from the initial memory image. During the second stage, the processor executes initialization code designed to build the remainder of the memory image so that execution of applications code can begin.

### First Stage of Initialization

The following procedure shows the steps that system hardware and the processor go through in the first stage of initialization. The algorithm in Figure 9-6 gives the details of this procedure.

1. Hardware asserts the RESET pin on the processor.
2. The processor samples LPN to get its local processor number (1 or 0). (LPN and STARTUP are signals that come from multiplexed information received on several processor pins.)
3. The processor asserts the FAILURE pin and performs a self test. If the processor passes the self test, it deasserts the FAILURE pin.
4. The processor samples STARTUP to determine whether it is the initializing processor (1) or not (0). If the processor is the initializing processor, it continues with the initialization procedure; if it is not, it goes into the stopped state. (In multiprocessing systems, all processors except the initializing processor are put in the stopped state.)
5. The processor reads the 8 check-sum words and checks that the check sum is 0.
6. Using the contents of the check-sum words, the processor determines the location of the initialization segment table, PRCB, and first instruction to be executed.
7. The processor sets its process priority to 31 (highest possible) and its state to idle interrupted.
8. The processor clears any latched external interrupt or IAC signals. This means that the processor will not service any interrupts or IACs prior to beginning instruction execution.
9. The processor begins executing the initialization instruction list.

```

assert  $\overline{\text{FAILURE}}$  pin;
perform self test;
if self test fails
    then enter stopped state;
else
    deassert  $\overline{\text{FAILURE}}$  pin;
    enter predefined state;
    if STARTUP pin = 0
        then enter stopped state;
    else
        x  $\leftarrow$  memory(0); read 8 words from
            physical address 0
        AC.cc  $\leftarrow$  0002;
        temp  $\leftarrow$  FFFFFFFF16 add_with_carry x(0);
        temp  $\leftarrow$  temp add_with_carry x(1);
        temp  $\leftarrow$  temp add_with_carry x(2);
        temp  $\leftarrow$  temp add_with_carry x(3);
        temp  $\leftarrow$  temp add_with_carry x(4);
        temp  $\leftarrow$  temp add_with_carry x(5);
        temp  $\leftarrow$  temp add_with_carry x(6);
        temp  $\leftarrow$  temp add_with_carry x(7);
        if temp  $\neq$  0
            then
                assert  $\overline{\text{FAILURE}}$  pin;
                enter stopped state;
            else
                segment_table_descriptor  $\leftarrow$ 
                    memory(x(0) + 128);
                IP  $\leftarrow$  memory(12)
                fetch PRCB;
                process.priority  $\leftarrow$  31;
                process.state  $\leftarrow$  interrupted;
                FP  $\leftarrow$  PRCB.interrupt_stack_pointer;
                clear any latched external interrupt/IAC
                    signals;
                begin execution;
            endif;
        endif;
    endif;
endif;

```

**Figure 9-6: Algorithm for First Stage of Initialization Procedure**

After self test, the processor establishes its initial state. For the initializing processor this state is idle; for any other processors in the system this state is stopped. Also at initialization, the trace controls are set to zero; the breakpoint registers are disabled; and the process controls are set to zero (except for the execution mode, which is set to supervisor, and the priority, which is set to 31).



When the initializing processor begins instruction execution, it goes into idle-interrupted state. The initialization code is thus treated as a special interrupt-handler procedure.

### **Second Stage of Initialization**

The processor activity during the second stage of initialization, which occurs once the processor begins instruction execution, is up to software. In general, this stage of initialization is used to copy or create additional data structures in memory, such as page tables, one or more PRCBs, the interrupt table, the system-procedure table, and the fault table (if not in the initial memory image).

To complete the initialization procedure, software will ordinarily bind a process to the processor to begin process execution. Refer to Chapter 13 for a full discussion of binding a process to a processor.

Once these jobs are completed, the processor can begin executing applications code.

Appendix D gives an example of the 80960MC code that might be used to carry out this second stage of initialization.

A common initialization technique is to create a new segment table and PRCB in memory along with the other system data structures that are placed in memory in the second initialization stage. The processor is then restarted using the new segment table and PRCB. (The code in Appendix D uses this technique.)

A processor is restarted using the restart IAC. The restart IAC message includes new physical-address pointers to the segment table and PRCB. The processor reads the new PRCB, then begins instruction execution according to the control information contained in the PRCB.

In a multiprocessing system, one of the processor's tasks following restart would be to complete the initialization of the other processors in the system. Further discussion of the initialization of multiprocessing systems is given in Chapter 15.



When the initializing processor begins instruction execution, it goes into idle-interrupted state. The initialization code is thus treated as a special interrupt-handler procedure.

## Second Stage of Initialization

The processor activity during the second stage of initialization, which occurs once the processor begins instruction execution, is up to software. In general, this stage of initialization is used to copy or create additional data structures in memory, such as page tables, one or more PRCBs, the interrupt table, the system-procedure table, and the fault table (if not in the initial memory image).

To complete the initialization procedure, software will ordinarily bind a process to the processor to begin process execution. Refer to Chapter 13 for a full discussion of binding a process to a processor.

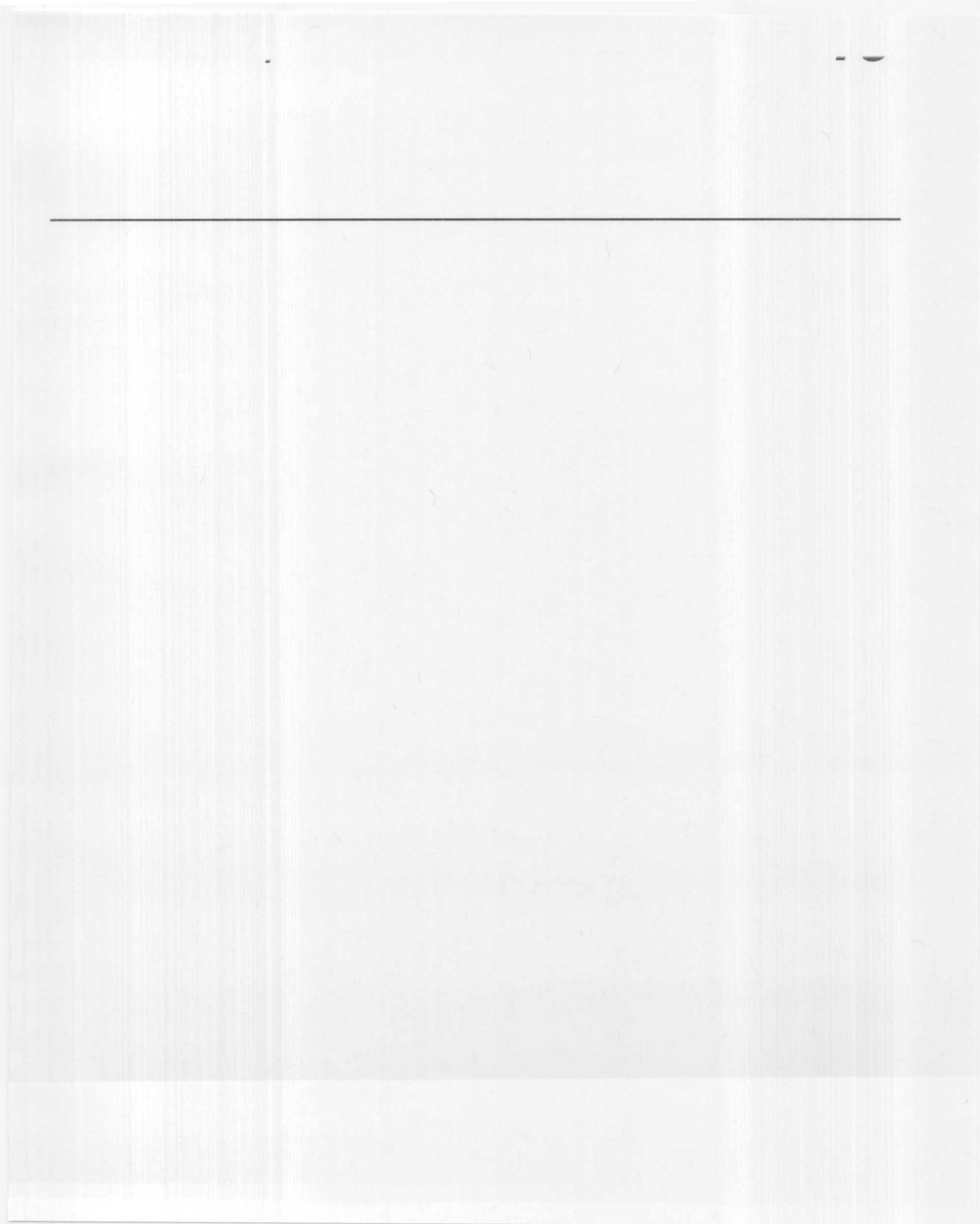
Once these jobs are completed, the processor can begin executing applications code.

Appendix D gives an example of the 80960MBC code that might be used to carry out this second stage of initialization.

A common initialization technique is to create a new segment table and PRCB in memory along with the other system data structures that are placed in memory in the second initialization stage. The processor is then restarted using the new segment table and PRCB. (The code in Appendix D uses this technique.)

A processor is restarted using the restart IAC. The restart IAC message includes new physical-address pointers to the segment table and PRCB. The processor reads the new PRCB, then begins instruction execution according to the control information contained in the PRCB.

In a multiprocessing system, one of the processor's tasks following restart would be to complete the initialization of the other processors in the system. Further discussion of the initialization of multiprocessing systems is given in Chapter 15.





## CHAPTER 10 INTERRUPTS

This chapter describes the 80960MC processor's interrupt handling facilities. It also describes how interrupts are signaled.

### OVERVIEW OF THE INTERRUPT FACILITIES

An interrupt is a temporary break in the control stream of a process so that the processor can handle another chore. Interrupts are generally requested from an external source. The interrupt request either contains a vector number or else points to a vector that tells the processor what chore to do while in the interrupted state. When the processor has finished servicing the interrupt, it generally returns to the process that it was last working on when the interrupt occurred and resumes execution where it left off.

The processor provides a mechanism for servicing interrupts, which uses an implicit procedure call to a selected interrupt-handling procedure, called an *interrupt handler*.

When an interrupt occurs, the current state of the process is saved. If the interrupt occurs during an instruction that requires many machine cycles, the instruction state is also saved and execution of the instruction is suspended.

The processor then creates a new frame on the interrupt stack and executes an implicit call to the interrupt handler selected with the interrupt vector.

Upon returning from the interrupt handler, the processor switches back to the process that was running when the interrupt occurred, restores this process to the state it was in when the interrupt occurred, and resumes work on the process.

Another feature of this interrupt handling mechanism is that it allows interrupts to be prioritized. If an interrupt is signaled that has the same or a lower priority than the process that the processor is currently working on, the processor saves the interrupt and services it at a later time. Interrupts that are waiting to be serviced are called *pending interrupts*.

### SOFTWARE REQUIREMENTS FOR INTERRUPT HANDLING

To use the processor's interrupt handling facilities, software must provide the following items in memory:

- Interrupt Table
- Interrupt Handler Routines
- Interrupt Stack

These items are generally established in memory as part of the initialization procedure. Once these items are present in memory and pointers to them have been entered in the appropriate system data structures, the processor then handles interrupts automatically and independently from software.

The requirements for these items are given in following sections of this chapter.

## VECTORS AND PRIORITY

Each interrupt vector is 8 bits in length, which allows up to 256 unique vectors to be defined. In practice, vectors 0 through 7 cannot be used, and vectors 244 through 247 and 249 through 251 are reserved and should not be used by software. Vector 248 is reserved for a processor generated interrupt called a *system-error interrupt*. This interrupt is described in Chapter 12 in the section titled "System-Error Interrupt."

Each vector has a predefined priority, which is defined by the following expression:

$$\text{priority} = \text{vector}/8$$

Thus, at each priority level, there are 8 possible vectors (e.g., vectors 8 through 15 have a priority of 1, vectors 16 through 23 a priority of 2, and so on to vectors 246 through 255, which have a priority of 31).

The processor uses the priority of an interrupt to determine whether or not to service the interrupt immediately or to delay service. If the interrupt priority is greater than the priority of the current process, the processor services the interrupt immediately; if the interrupt priority is equal to or lower than the priority of the current process, the processor saves the interrupt vector as a pending interrupt so that it can be serviced after work on the current process is complete.

A priority-31 interrupt is always serviced immediately.

Note that the lowest process priority allowed is 0. If the current process has a 0 priority, a priority-0 interrupt will never be accepted. This is why vectors 0 through 7 cannot be used. In fact, there are no entries provided for these vectors in the interrupt table.

## INTERRUPT TABLE

The interrupt table contains instruction pointers (addresses in the address space) to interrupt handlers. This table is located in physical memory and must be aligned on a word boundary. The processor determines the location of the interrupt table by means of a physical address pointer in the PRCB.

As shown in Figure 10-1, the interrupt table contains one entry (i.e., one pointer) for each allowable vector. The structure of an interrupt-table entry is given at the bottom of Figure 10-1. Each interrupt procedure must begin on a word boundary, so the two least-significant bits of the entry are set to 0.





The instruction pointers can be either physical or virtual addresses. Which kind of address is used depends on the address-translation mode that the processor is set for: physical addressing or virtual addressing.

The first 36 bytes of the interrupt table are used to record pending interrupts. This section of the table is divided into two fields: pending priorities (byte-offset 0 through 3) and pending interrupts (byte-offset 4 through 35).

The pending priorities field contains a 32-bit string in which each bit represents an interrupt priority. The bit number in the string represents the priority number. When the processor posts a pending interrupt in the interrupt table, the bit corresponding to the interrupt's priority is set. For example, if an interrupt with a priority of 10 is posted in the interrupt table, bit 10 is set.

The pending interrupts field contains a 256-bit string in which each bit represents an interrupt vector. For example, byte-offset 4 is reserved, byte-offset 5 is for vectors 8 through 15, byte-offset 6 is for vectors 16 through 23, and so on. When a pending interrupt is logged, its corresponding bit in the pending interrupt field is set.

This encoding of the pending priority and pending interrupt fields permits the processor to first check if there are any pending interrupts with a priority greater than the current process and then to determine the vector number of the interrupt with the highest priority. Software should set these fields to 0 at initialization and not access these fields after that.

### INTERRUPT-TABLE SHARING

One of the reasons that the interrupt table is located in physical memory is to enable systems that use multiple processors to share the interrupt table. Then when one processor receives an interrupt and posts it as a pending interrupt in the interrupt table, another processor can service the interrupt. Refer to the section in Chapter 15 titled "Interrupt Handling in a Multiprocessor System" for further information on interrupt table sharing.

### INTERRUPT-HANDLER PROCEDURES

An interrupt handler is a procedure that is designed to perform a specific action that has been associated with a particular interrupt vector. For example, a typical job for an interrupt handler is to read a character from a keyboard.

#### Location of Interrupt Handler

The interrupt handler procedures can be located in physical memory or virtual memory, depending on the address-translation mode to be used. If the procedures are located in virtual memory, they are generally mapped in region 3 of the address space so that they are available to all processes. As stated in the previous section, each procedure must begin on a word boundary.

## Interrupt-Handler Restrictions

The processor execution mode is always switched to supervisor while an interrupt is being handled. The pages that contain interrupt handler routines may thus have their page rights set for supervisor only access.

When an interrupt-handler procedure is called, the states of the process controls and arithmetic controls for the interrupted process are saved. However, the interrupt handler shares the other resources of the interrupted process, in particular the global registers and the address space. This sharing of resources imposes two important restrictions on the interrupt handler procedures.

First, the interrupt handler procedures must preserve and restore the state of any of the resources that it uses. For example, the processor allocates a set of local registers to the interrupt handler, just as it does on a local procedure call. If the interrupt handler needs to use the global or floating-point registers, however, it should save their contents before using them and restore them before returning from the interrupt.

Second, the interrupt handler should not do anything that would cause the interrupted process to be unbound from the processor and rescheduled, because doing so would leave the processor in an indeterminate state. To avoid rescheduling the process, an interrupt handler should not use the **sendserv** (send service), **receive**, and **wait** instructions. Also, the interrupt handler should not enable timing (set the timing flag in the process controls register), since this can result in an end-of-time-slice event that can also cause the interrupted process to be rescheduled.

The **resumpres** instruction (resume process) can be used; however, the state of the interrupted process will be lost.

An interrupt-handler procedure can also be called when the processor is not currently executing a process. One example of this situation is when the processor receives an interrupt while it is servicing another interrupt. Here, execution of the **ldtime** instruction (load process time) or the **condrec** instruction (conditional receive) returns an undefined result.

## INTERRUPT STACK

The interrupt stack is usually located in region 3 of the address space. The processor determines the location of the interrupt stack by means of a pointer in the PRCB. To avoid raising a fault while processing an interrupt, the interrupt stack must be frozen in physical memory, meaning that the pages that contain the stack must always be valid.

The interrupt stack has the same structure as the local procedure stack described in Chapter 4 in the section titled "Procedure Stack."

## PROCESS TIMING WHILE HANDLING AN INTERRUPT

When an interrupt occurs while the processor is executing a process, the processor stops counting process time (i.e., stops counting down the residual-time-slice value) while it is executing the interrupt-handler procedure. Thus, the time required to handle an interrupt is not counted as part of the process's time slice.

## SIGNALING INTERRUPTS

The processor can be interrupted in any of the following six ways:

- Signal on its interrupt pins
- Signal on its interrupt pins from an external interrupt controller
- An IAC message from external source
- An IAC message from a program in the processor
- A system-error fault interrupt
- A pending interrupt (described at the end of the chapter)

### Interrupts From Interrupt Pins

The processor has four interrupt pins, called INT0, INT1, INT2, and INT3. These pins can be configured in either of the following three ways:

- as four interrupt-signal inputs;
- as two interrupt inputs and two pins for handshaking with an interrupt controller such as the Intel M8259A Programmable Interrupt Controller; or
- as one IAC input and three interrupt inputs.

A 32-bit, interrupt-control register in the processor determines how these pins are used. Each interrupt pin is associated with one 8-bit field in the register, as shown in Figure 10-2.

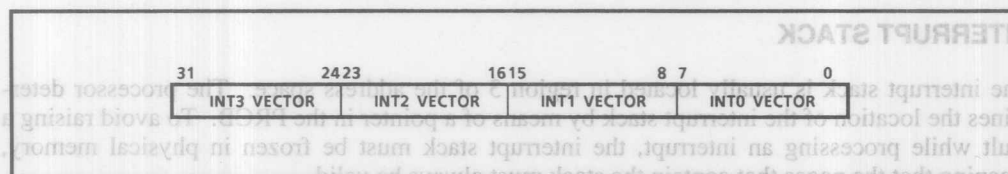


Figure 10-2: Interrupt-Control Register

If the interrupt pins are to be used as four inputs, a different interrupt vector is stored in each of the four fields in the interrupt-control register. Then when an interrupt is signaled on one of the pins, the processor reads the vector from the pin's associated field in the register. For example, if an interrupt is signaled on pin INT0, the processor reads the vector from bits 0 through 7.

The processor assumes that the interrupt vectors in the interrupt register are arranged in descending order from the INT0 field to the INT3 field (e.g., the priority of  $\text{INT0} \geq \text{INT1} \geq \text{INT2} \geq \text{INT3}$ ). To insure that interrupts are handled in the proper order, software should follow this convention.

If the INT0 vector field is set to 0, the function of the INT0 pin is changed to IAC, and it is used to signal the processor that an external IAC message has been sent to it. In fact, the INT0 pin must be configured in this manner for the processor to service external IAC messages.

If the INT2 vector field is set to 0, the functions of the INT2 and INT3 pins are changed to INTR and INTA, respectively. Here, the INTR pin is used to receive signals from an interrupt controller and the INTA pin is used to send acknowledge signals back to the controller. When the processor receives a signal on the INTR pin, it reads an interrupt vector from the least-significant 8 bits of the local bus, then sends an acknowledge signal to the controller through INTA. When the INT2 and INT3 pins are configured in this manner, the processor ignores the INT3 vector field.

The interrupt-control register is memory mapped to physical addresses  $\text{FF000004}_{16}$  through  $\text{FF000007}_{16}$ . Only the processor can read or write this register using the synchronous load (**synld**) and synchronous move (**synmov**) instructions. External agents on the bus cannot access this register.

#### NOTE

If the virtual-addressing mode is going to be used, a page in region 3 will need to be mapped to the page in the physical address space that contains the addresses ranging from  $\text{FF000004}_{16}$  through  $\text{FF000007}_{16}$ . Software can then read from or write to the interrupt control register by referencing the addresses in region 3 that are mapped to the physical addresses of the register.

The value in the interrupt-control register after the processor is initialized is  $\text{FF000000}_{16}$ . With this setting, interrupt pin INT0 is used to signal an IAC; INT1 is inactive; and INT2 and INT3 are configured to perform handshaking with an interrupt controller.

### IAC Interrupts

The processor can also receive an interrupt request by means of the IAC mechanism. (The IAC mechanism is described in detail in Chapters 11 and 15.) The interrupt IAC message can be sent to the processor either from an external bus agent, such as an I/O processor or another CPU, or internally as part of the currently running process. The interrupt vector is contained in the interrupt IAC message.

As with any other IAC message, the processor receives notice of an external interrupt-IAC message through the INT0 pin, which has been configured as an IAC pin, as described in the previous section. The processor then reads the IAC message to get the interrupt vector.

A program running on the processor can signal an interrupt through an internal interrupt-IAC message. An internal IAC is sent to the processor by means of a synchronous move instruction. When the processor executes a synchronous move to its IAC message space, it signals an IAC message internally. The processor then reads the IAC message as it would for an external IAC.

### System-Error Interrupt

Under certain conditions, a system-error interrupt is signaled internally in the processor. This interrupt causes an explicit call to interrupt vector 248. The system-error interrupt mechanism, action, and possible handling methods are described in Chapter 12 in the section titled "System-Error Interrupt Action."

## INTERRUPT-HANDLING ACTIONS

As was described earlier in this chapter, when the processor receives an interrupt, it handles it automatically. The processor takes care of saving the process state, calling the interrupt-handler routine, and restoring the process state once the interrupt has been serviced. Software support is not required.

The following section describes the actions the processor takes while handling interrupts. It is not necessary to read this section to use the interrupt mechanism or write an interrupt handler routine. This discussion is provided for those readers who wish to know the details of the interrupt handling mechanism.

### Receiving an Interrupt

Whenever the processor receives an interrupt signal, it performs the following action:

1. It temporarily stops work on its current job, whether it is working on a process or another interrupt handler procedure.
2. It reads the interrupt vector from the interrupt register, the bus, or the IAC message space.
3. It compares the priority of the vector with the priority of the current process or the interrupt it is currently handling.
4. If the priority of the new interrupt is higher than that of the current process or interrupt, the processor services the new interrupt immediately as described in the next sections.
5. If the interrupt priority is equal to or less than that of the current process or interrupt, the processor records new interrupt in the pending interrupt record and continues work on the current process or interrupt.

### Servicing an Interrupt

The method that the processor uses to service an interrupt depends on the state the processor is in when it receives the interrupt. The following sections describe the interrupt handling actions for various states of the processor. In all of these cases, it is assumed that the interrupt is a higher priority than the current process and will thus be serviced immediately after the processor receives it. The handling of lower priority interrupts is described later in this chapter in the section titled "Servicing a Pending Interrupt."



### Process-Executing-State Interrupt

When the processor receives an interrupt while it is in the process-executing state, it performs the following actions to service the interrupt; this procedure is the same regardless of whether the processor is in the user or the supervisor mode when the interrupt occurs:

1. The processor switches to the interrupt stack (as shown in Figure 10-3). The interrupt stack pointer becomes the new stack pointer (NSP) for the processor.
2. The processor saves the current state of process controls and arithmetic controls in an interrupt record on the interrupt stack. (The interrupt record is described later in this chapter in the section titled "Interrupt Record".)
3. If the execution of an instruction was suspended, the processor includes a resumption record for the instruction in the interrupt record and sets the resume flag in the saved process controls. (Refer to the section in Chapter 9 titled "Instruction Suspension" for a discussion of the criteria for suspending instructions.)
4. The processor allocates a new frame on the interrupt stack and loads the new frame pointer (NFP) in global register g15.
5. The processor switches to the process-interrupted state.
6. The processor sets the process state flag in its internal process controls to interrupted, its execution mode to supervisor, and its priority to the priority of the interrupt. Setting the processor's priority to that of the interrupt insures that lower priority interrupts can not interrupt the servicing of the current interrupt.
7. Also in the current process controls, the processor clears the trace-fault-pending, timing, trace-enable, and time-slice flags. Clearing these flags allows the interrupt to be handled without trace faults being raised and without the process timing out.
8. The processor sets the frame return status field (associated with the PFP in r0) to 111<sub>2</sub>.
9. The processor performs an implicit call-extended operation (similar to that performed for the `callx` instruction). The address for the procedure that is called is that which is specified in the interrupt table for the specified interrupt vector.

Once the processor has completed the interrupt procedure, it performs the following action on the return:

1. The processor copies the arithmetic controls field from the interrupt record into its arithmetic controls register.
2. The processor copies the process controls field from the interrupt record into its internal process controls.
3. If the resume flag of the process controls is set, the processor copies the resumption record from the interrupt record to the resumption record field of the PCB for the process being resumed.
4. The processor deallocates the current stack frame and interrupt record from the interrupt stack and switches to the local stack or the supervisor stack (whichever one it was using when it was interrupted).
5. The processor checks the interrupt table for pending interrupts that are higher than the priority of the process being returned to. If a higher-priority pending interrupt is found, it is handled as if the interrupt occurred at this point.



6. Assuming that there are no pending interrupts to be serviced, the processor switches to the process-executing state and resumes work on the current process.

If the processor is configured to use the high-level process management facilities or multiple processors or both, the processor performs the following additional operations prior to resuming work on the interrupted process:

1. If either the multiprocessor-preempt flag or the check-dispatch-port flag in the processor controls is set, the processor checks the dispatch port and clears the check-dispatch-port flag. Otherwise, it goes to step 4.
2. If the dispatch port contains a process whose priority is higher than that of both the current process and the value in the nonpreempt-limit field in the processor controls, the processor suspends the current process and enqueues it at the front of the queue for its associated dispatch port. The processor then dispatches the higher priority process, which becomes the current process.
3. If a higher priority process was not found on the dispatch port, the process that was interrupted remains the current process.
4. The processor then begins work on the current process.

### Process-Interrupted-State Interrupt

If the processor receives an interrupt while it is servicing another interrupt, and the new interrupt has a higher priority than the interrupt currently being serviced, the current interrupt-handler routine is interrupted. Here, the processor performs the same action to save the state of the current interrupt-handler routine as is described at the beginning of this section. The interrupt record is saved on the top of the interrupt stack, prior to the new frame that is created for use in servicing the new interrupt.

On the return from the current interrupt handler to the previous interrupt handler, the processor deallocates the current stack frame and interrupt record, and stays on the interrupt stack.

### Interrupt Record

The processor saves the state of the interrupted process in an interrupt record. Figure 10-3 shows the structure of this interrupt record. The resumption record within the interrupt record is used to save the state of a suspended instruction. If no instruction is suspended, the resumption record is not created.

### Idle-State Interrupt

The processor can also be interrupted while in the idle state. The processor handles such interrupts in essentially the same way that it handles interrupts that occur while the processor is in the process-executing state, with the following exception. When the processor allocates the new frame on the interrupt stack, it sets the frame return field to 110<sub>7</sub>. This causes the processor to revert to the idle state when the processor returns from the interrupt-handler procedure.



### Idle-Interrupted State Interrupt

If the processor receives an interrupt while it is in the idle-interrupted state, it handles the interrupt just as it would if it occurred in the process-interrupted state.

### Pending Interrupts

As is described earlier in this chapter, the processor provides a mechanism for evaluating interrupts according to their priority. If the interrupt priority is equal to or lower than the priority of the current process, the processor does not service the interrupt immediately. Instead, it posts the interrupt in the pending interrupt section of the interrupt table. The processor checks the interrupt table at specific times and services those interrupts that have a higher priority than its current priority. This pending interrupt mechanism provides two benefits:

1. The ability to delay the servicing of low priority interrupts (by posting them in the pending interrupt section of the interrupt table) allows the processor to concentrate its processing activity on higher priority tasks.
2. In a system that uses two or more 80960MC processors, both processors can share the same interrupt table. This interrupt-table sharing allows the processors to share the interrupt handling load.

The following paragraphs describe how the processor handles pending interrupts.

#### NOTE

The 80960 architecture defines the section of the interrupt table for storing pending interrupts and a mechanism for checking the interrupt table for pending interrupts. The method used for posting interrupts to the interrupt table and circumstances under which the processor check the interrupt table for pending interrupts is not defined.

In the following description of the pending interrupt mechanism, the information given in the sections titled "Posting Pending Interrupts" and "Checking for Pending Interrupts" is specific to the 80960MC processor. The information given in the section titled "Handling Pending Interrupts" is defined in the 80960 architecture and should be common in all processors that implement this part of the architecture.

### Posting Pending Interrupts

An interrupt can be posted in the pending-interrupt record of the interrupt table in either of the following two ways:

1. The processor receives an interrupt with a priority equal to or lower than that of the process the processor is currently working on. The processor then automatically posts the interrupt in the pending-interrupt record.
2. The kernel can set the desired pending-interrupt and pending-priority bits in the interrupt table.

Using the first method, the processor performs an atomic read/write operation that locks the interrupt table until the posting operation has been completed. Locking the interrupt table prevents other agents on the bus from accessing the interrupt table during this time.

The second method of posting an interrupt is risky, because it does not use this locking technique. (The processor's atomic instructions are not able to perform a locking operation that spans several instructions.) This method will work only if the kernel can insure the following:

- that no external I/O agent will attempt to post a pending interrupt simultaneously with the processor, and
- that an interrupt cannot occur after one bit (e.g., the pending priority bit) of the pending-interrupt record is set but before the other bit (the pending interrupt vector) is set.

### Checking for Pending Interrupts

The processor automatically checks the interrupt table for pending interrupts at the following times:

- After returning from an interrupt-handler procedure
- While executing a modify-process-controls instruction (**modpc**), if the instruction causes the process's priority to be lowered.
- After receiving a test pending interrupts IAC message.

### Handling Pending Interrupts

The processor uses the same type of atomic read/write operation to check the interrupt table for pending interrupts as it does for posting pending interrupts. Again, this technique prevents other agents on the bus from accessing the interrupt table until the pending-interrupt check has been completed.

When the processor finds a pending interrupt, it handles it as if it had just received the interrupt. The handling mechanism is the same as is described earlier in this chapter for interrupts that are serviced as soon as they are received.

If the processor finds two pending interrupts at the same priority, it services the interrupt with the highest vector number first.

The second method of posting an interrupt is risky, because it does not use this locking technique. (The processor's atomic instructions are not able to perform a locking operation that spans several instructions.) This method will work only if the kernel can insure the following:

- that no external I/O agent will attempt to post a pending interrupt simultaneously with the processor, and
- that an interrupt cannot occur after one bit (e.g., the pending priority bit) of the pending-interrupt record is set but before the other bit (the pending interrupt vector) is set.

### Checking for Pending Interrupts

The processor automatically checks the interrupt table for pending interrupts at the following times:

- After receiving a test pending interrupt IAC message.
- While executing a modify-process-controls instruction (mopc), if the instruction causes the process's priority to be lowered.
- After returning from an interrupt-handler procedure.

### Handling Pending Interrupts

The processor uses the same type of atomic read/write operation to check the interrupt table for pending interrupts as it does for posting pending interrupts. Again, this technique prevents other agents on the bus from accessing the interrupt table until the pending-interrupt check has been completed.

When the processor finds a pending interrupt, it handles it as if it had just received the interrupt. The handling mechanism is the same as is described earlier in this chapter for interrupts that are serviced as soon as they are received.

If the processor finds two pending interrupts at the same priority, it services the interrupt with the highest vector number first.

---

*Interagent Communication* **11**

---





## CHAPTER 11 INTERAGENT COMMUNICATION

This chapter describes the interagent communication (IAC) mechanism for the 80960MC processor. Included is a description of the IAC-message structure, the internal-IAC-message sending and receiving mechanism, and reference information on the available IAC messages.

The mechanism for sending and receiving external-IAC messages is described in Chapter 15.

### INTRODUCTION TO IAC MESSAGES

The IAC facilities provide a mechanism for agents on the local bus or AP bus to communicate with one another by means of messages. The agents that use these facilities are primarily CPU processors such as the 80960MC and I/O processors. However, special processors that have a need to communicate with the other processors in the system may also use the IAC facilities.

The primary function of these facilities is to give multiple processors within a system a simple means of coordinating their activities. This capability is particularly important when the processors share a common memory space.

The IAC facilities are also used in single-processor systems for functions such as changing the processor's state or updating address-translation information.

IAC messages (referred to here as IACs) are four words in length and are exchanged by means of message buffers that are mapped to physical memory. All the usable IACs are predefined. The processor handles an IAC in much the same way as it handles an instruction.

The processor provides two mechanisms for exchanging IACs: external and internal. The external IAC mechanism is used to pass IACs between two agents, either on the local bus or on the AP bus. A processor uses the internal IAC mechanism to pass an IAC to itself.

This chapter describes the internal IAC mechanism, which is the only mechanism used in single-processor systems. The external IAC mechanism is described in Chapter 15 in the section titled "External IAC Message Passing."

### SOFTWARE REQUIREMENT FOR HANDLING INTERNAL IACS

No special software, such as dedicated data structures or stacks, are required to handle internal IACs. An internal IAC is sent with a quad synchronous move instruction (**synmovq**). The processor receives and handles the IAC internally.

SUMMARY OF IAC MESSAGES

Table 11-1 gives a list of the IAC messages that the processor can send either internally or externally. The messages marked with an asterisk are generally not used with single-processor systems. Detailed reference information on these messages is given at the end of this chapter.

Table 11-1: IAC Messages

<b>Interrupt Handling</b>	<b>Process Management</b>
Interrupt	Flush Local Registers
Test Pending Interrupt	Flush Process
	Preempt Process*
<b>Processor Management</b>	Purge Instruction Cache
Store System Base	Set Breakpoint Register
Store Processor	Check Process Notice*
Modify Processor Controls	
Stop Processor*	<b>Memory Management</b>
Freeze*	Flush TLB Physical Page
Restart Processor	Flush TLB
Warmstart Processor	Flush TLB Segment Entry
Continue Initialization	Flush TLB Page Table Entry
Reinitialize Processor	

IAC-MESSAGE FORMAT

Figure 11-1 shows the format for an IAC message. Each message is four words in length and consists of a message-type field and up to five parameter fields.

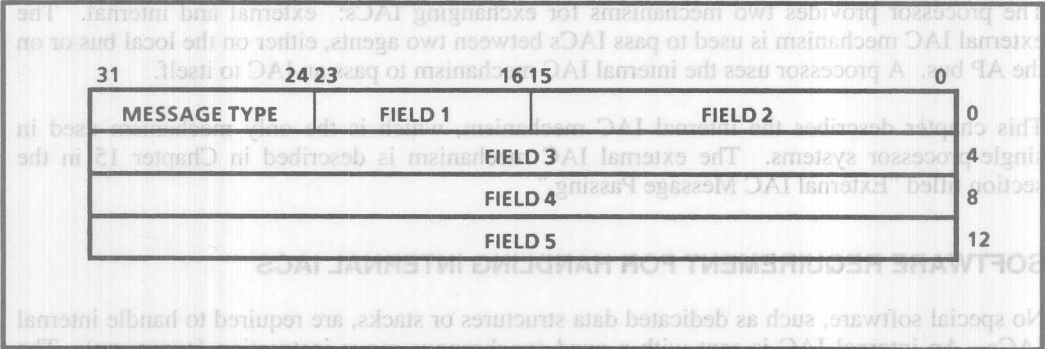


Figure 11-1: IAC-Message Format

The message type is an 8-bit binary code. Each IAC has a unique message type. The parameters can be 8, 16, or 32-bits in length, depending on the specified field. Many of the IACs do not require parameters. When a message type does require one or more parameters, the processor only looks at the required parameter fields. Those fields not used are ignored.

## SENDING AND RECEIVING AN INTERNAL IAC

To send an internal IAC, software must perform the following steps:

1. Load the message into four consecutive words in memory, with the first word aligned on a word boundary.
2. Execute a **synmovq** instruction to move the message from its source address to the destination address  $FF000010_{16}$ , where  $FF000010_{16}$  is a physical address.

When the destination operand of a **synmovq** instruction is  $FF000010_{16}$ , the processor interprets the instruction as a send internal-IAC instruction. The processor then receives the IAC by moving the message from memory into an internal message buffer.

The action of the **synmovq** move instruction insures that the loading of the message into the processor is completed before the processor is allowed to perform any other chores.

### NOTE

If the virtual-addressing mode is going to be used, a page in region 3 will need to be mapped to the page in the physical address space that contains address  $FF000010_{16}$ . Software can then send an internal IAC by writing to the address in region 3 that is mapped to physical address  $FF000010_{16}$ .

## INTERNAL-IAC-HANDLING ACTION

All internal IACs are assumed to have a priority of 31, so the processor executes the action requested in the IAC message immediately, even if the processor is currently working on a process or interrupt with a priority of 31.

The processor handles IACs internally. It does not use any of the resources of the execution environment such as the registers (global or local), the stack, or memory. Thus, the state of the process or processor when the IAC is received does not need to be saved.

Some IACs, such as the flush TLB IACs, do not affect the process or processor state. The processor treats these IACs as if they were an instruction inserted in the control flow of the process. When the IAC action is complete, the processor resumes work on the current process.

Other IACs, such as the restart processor and preemption IACs, cause the state of the processor or the control of the current process to be permanently changed. In these instances, the processor resumes activity in its new processor state or process state or both, following the execution of the IAC.

While the processor is handling an IAC, it will not respond to interrupts signaled on the interrupt pins.

## IAC FAULTS

If a fault condition occurs during the handling of an IAC message, a structural IAC fault is signaled. If when a structural IAC fault occurs, the processor is in the process-executing state,

the fault is handled within the environment of the current process. If the processor is not in the process-executing state, the fault is handled by means of a system-error interrupt.

## IAC-MESSAGE REFERENCE

The following section provides detailed descriptions of the operations carried out for each of the IACs. This section is organized alphabetically by IAC title for easy reference.

### NOTE

If the virtual-addressing mode is going to be used, a page in region 3 will need to be mapped to the page in the physical address space that contains address FF000010<sub>16</sub>. Software can then send an internal IAC by writing to the address in region 3 that is mapped to physical address FF000010<sub>16</sub>.

## INTERNAL-IAC-HANDLING ACTION

All internal IACs are assumed to have a priority of 31, so the processor executes the action requested in the IAC message immediately, even if the processor is currently working on a process or interrupt with a priority of 31.

The processor handles IACs internally. It does not use any of the resources of the execution environment such as the registers (global or local), the stack, or memory. Thus, the state of the process or processor when the IAC is received does not need to be saved.

Some IACs, such as the flush TLB IACs, do not affect the process or processor state. The processor treats these IACs as if they were an instruction inserted in the control flow of the process. When the IAC action is complete, the processor resumes work on the current process.

Other IACs, such as the restart processor and preemption IACs, cause the state of the processor or the control of the current process to be permanently changed. In these instances, the processor resumes activity in its new processor state or process state or both, following the execution of the IAC.

While the processor is handling an IAC, it will not respond to interrupts signaled on the interrupt pins.

## IAC FAULTS

If a fault condition occurs during the handling of an IAC message, a structural IAC fault is signaled. If when a structural IAC fault occurs, the processor is in the process-executing state,

## Check Process Notice

**Message Type:**

90<sub>16</sub>

**Parameters:**

Fields 1-2

Not Used

Field 3

SS of PCB

Fields 4 - 5

Not Used

**Function:**

Examines the process-notice field of the PCB for the current process. If the event-fault-request flags in this field are set, the flags are cleared and an event-notice fault is signaled. Otherwise, no action is taken.

The field 3 parameter contains the SS of the PCB. When the processor receive this IAC, it checks this parameter for either of the following conditions: (1) the field is zero or (2) the field contains the SS for the current process PCB. If either of these conditions is true and the process is not in an interrupted state, the processor checks the uncached process-notice field from the PCB in memory, as described above. If neither condition is true, no action is taken.



## Continue Initialization

**Message Type:**

92<sub>16</sub>

**Function:**

Carries out the initialization procedure that follows the processor self test. If the processor is the initializing processor, it puts itself in the idle state and executes the initialization procedure beginning with reading the initial memory image from ROM. The self test is not performed.

If the processor is not the initializing processor, it puts itself in the stopped state and no further action is performed.

Refer to the section in Chapter 12 titled "Processor Initialization" for further details on the initialization process.

## Flush Local Registers

**Message Type:**84<sub>16</sub>**Parameters:**

Fields 1- 2 Not Used

Field 3

Physical Address of Stack Page

Fields 4- 5

Not Used

**Function:**

Writes the contents of the all local-register sets (located in the on-chip local-register cache) to their associated stack frames in memory. The field 3 parameter contains the base physical-address of a page that contains all or part of the stack to be written to. If any of the local register sets are associated with a stack frame in the specified page, all of the local register sets are flushed to memory. Then, all the register sets except the current set (the set for the active frame) are marked as purged. This means that on a return to a register set that has been purged, the processor will load these registers from the stack.

No action is taken if (1) none of the register sets are associated with a stack frame in the specified page or (2) the processor is in the stopped or idle state.

## Flush Process

**Message Type:**87<sub>16</sub>**Function:**

Suspends the current process, then rebinds the processor to that process. This action is carried out only if the processor is in the process-executing state. Since the process is literally suspended and rebound, process timing is turned off then back on again as a result of this action.

This IAC also causes the following items to be invalidated in the TLB: the segment descriptor for the current PCB, the segment descriptors for regions 0, 1, and 2 for the current process; and the page-table entries for pages addressed by addresses in regions 0, 1, and 2.

If the processor is not in the process-executing state, no action is taken.

**Flush TLB****Message Type:**8A<sub>16</sub>**Function:**

Invalidates all TLB entries except the following: (1) the segment descriptors for the segment-table and region 3, (2) the segment descriptor for the current process, (3) the segment descriptors for regions 0, 1, and 2 of the current process, and (4) the page-table entry for the page in which the interrupt stack begins.

## Flush TLB Page Table Entry

**Message Type:**

8C<sub>16</sub>

**Parameters:**

Fields 1 - 2 Not Used

Field 3

Offset From Segment Base

Field 4

SS of Segment That Contains Page

Field 5

Not Used

**Function:**

Invalidates the page-table entry for the page specified with the field 3 and field 4 parameters. The processor determines the page that contains the address specified by the SS and offset in fields 4 and 3, respectively. If a TLB entry exists for this page, the processor flushes the entry.

This IAC can generate a protection-length fault if the specified address is beyond the specified length of the segment.

Note that field 3 is not interpreted as an address within the address space, but as an offset within a segment. Thus, to flush an entry for a page in an address space that contains a particular address, the following steps must be taken. (1) The SS for the region that contains the address is supplied in field 4. (2) The two most-significant bits of the address are cleared to form the offset into the region. This offset is then supplied in field 3.

This IAC should not be used to flush page-table-directory entries, because they are never saved in the TLB.

**Flush TLB Physical Page****Message Type:**88<sub>16</sub>**Parameters:**

Fields 1 - 2

Not Used

Field 3

Base Physical Address of Page

Fields 4 - 5

Not Used

**Function:**

Invalidates all the entries in the TLB that point directly to the page specified with the field 3 parameter. The entries that may be flushed with this IAC include (1) segment descriptors and page-table entries that point to the page, (2) the segment descriptors for paged segments that point to a page table in that page, and (3) the segment descriptors for bipaged segments that point to a page-table directory in that page.

Also, the function of the flush-local-registers IAC message is performed. And, if the segment descriptor for the PCB of the current process or the segment descriptors for regions 0, 1, or 2 of the current process are invalidated, the function of the flush-process IAC message is performed.

Note that this function is slower than the flush functions of the other IAC messages. However, the function that this IAC performs is needed for situations where processes share pages.



# Flush TLB Segment Entry

Message Type:

8B<sub>16</sub>

Parameters:

Fields 1 - 2 - Not Used

Fields 1 - Not Used

Field 3 - Base Physical Page

SS for Segment

Fields 4 - 5 - Not Used

Fields 4 - Not Used

**Function:** Invalidates all entries in the TLB that pertain to the segment specified in the field 3 parameter. The entries that may be flushed include (1) any segment-descriptor entry for the segment and (2) any associated page-table entries.

Note that this function is slower than the flush functions of the other IAC messages. However, the function that this IAC performs is needed for situations where processes share pages. Also, the function of the flush-local-register IAC message is performed. And, if the segment descriptor for the PCB of the current process or the segment descriptors for regions 0, 1, or 2 of the current process are invalidated, the function of the flush-process IAC message is performed.

**Freeze****Message Type:**91<sub>16</sub>**Function:**

Stops the processor without suspending the current process. The processor puts itself in the stopped state. If the processor is in the process-executing state when this IAC is received, the current process is not suspended.

## Interrupt

**Message Type:**40<sub>16</sub>**Parameters:**

Field 1 Interrupt vector

Fields 2 - 5 Not Used

**Function:**

Generates an interrupt request. The interrupt vector is given in field 1 of the IAC message. The processor handles the interrupt request just as it does interrupts received from other sources. If the interrupt priority is higher than the priority of the current process, the processor services the interrupt request immediately. Otherwise, it posts the interrupt in the pending interrupts section of the interrupt table.

Refer to Chapter 10 for further information on the servicing of interrupt IACs.



Message Type:

8D<sub>16</sub>

Parameters:

Fields 1-2

Not Used

Field 3

New Processor Controls Word

Field 4

Mask

Field 5

Not Used

Function:

Modifies the processor controls word in the PRCB according to the *new value* given in field 3 and under control of the *mask* given in field 4. The mask determines which bits of the processor controls word may be changed according to the following relationship:

$\text{processor\_controls\_word} \leftarrow (\text{new value and mask})$

$\text{or } (\text{processor\_controls\_word and not } (\text{mask}))$

If any parts of the processor-controls word have been cached on the chip, they are updated as a result of this operation, with the exception of the processor-state bits. To explicitly change the state of the processor, the processor must be restarted (using the restart IAC) in the new state.

Refer to the section in Chapter 9 titled "Changing the Address-Translation Mode" for information on the effects of using the modify processor controls IAC to change the address-translation-mode flag.

## Preempt Process

**Message Type:**

85<sub>16</sub>

**Function:**

Suspends the current process and binds the processor to a higher priority process from the dispatch port. If the processor is in the idle or process-executing state, it checks the queue status field of the dispatch port. If the processor finds a process with a higher priority than that of both the current process and the nonpreempt-limit in the process controls, the processor performs the preemption action.

No action is taken if (1) the processor is in the stopped or an interrupted state, or (2) the priority of the highest priority process on the dispatch port is less than that of the current process or the nonpreempt-limit. More information on process preemption is given in Chapter 14 in the section titled "Process Preemption" and in Chapter 15 in the section "Multiprocessor Process Preemption."

Purge Instruction Cache

Message Type: 89<sub>16</sub>  
Function: Invalidates all entries in the processor's internal instruction cache.

Field 3 Address of System Address Table  
Field 4 Address of Processor Control Block  
Field 5 Start Instruction IP  
Function: Reestablishes the processor state. In reinitializing itself, the processor first locates the system address table and the processor control block in the IMI from the addresses given in fields 3 and 4. The processor then begins executing the instruction list beginning with the IP given in field 5.



**Reinitialize Processor****Message Type:**93<sub>16</sub>**Parameters:**

Fields 1- 2

Not Used

Field-3

Address of System Address Table

Field-4

Address of Processor Control Block

Field 5

Start Instruction IP

**Function:**

Reestablishes the processor state. In reinitializing itself, the processor first locates the system address table and the processor control block in the IMI from the addresses given in fields 3 and 4.

The processor then begins executing the instruction list beginning with the IP given in field 5.

## Restart Processor

Message Type:

81<sub>16</sub>

Parameters:

Fields 1 - 2

Not Used

Field-3

Physical Address of Segment Table

Field-4

Physical Address of PRCB

Field 5

Not Used

**Function:** Reestablishes the processor state. In restarting itself, the processor first locates the segment table and PRCB from the base physical addresses given in fields 3 and 4. (Field 3 is only used to locate the eighth segment-table entry, which is used thereafter to locate the segment table.)

Next, the processor checks the state field in the processor-controls word in the PRCB and enters that state. If the PRCB state is process-executing, the processor performs a bind action on the process whose SS is in the current-process-SS field in the PRCB.

## Set Breakpoint Register

Message Type:

8F<sub>16</sub>

Parameters:

Fields 1 - 2 Not Used

Not Used - Fields 1 - 2

Field 3 Physical Address Segment Table

Breakpoint IP - Field 3

Field 4 Physical Address PRCB

Breakpoint IP - Field 4

Field 5 Not Used

Not Used - Field 5

**Function:** Enables or disables two breakpoints. When the processor receives this IAC, it conditionally loads the parameters from fields 3 and 4 into breakpoint registers 0 and 1, respectively. Field 3 provides a breakpoint IP for breakpoint register 0, and field 4 provides a breakpoint IP for breakpoint register 1. Bit 1 in each of these fields is a breakpoint-disable flag.

If the disable flag in one of these fields is set, the breakpoint for the corresponding breakpoint register is disabled. Otherwise, the IP value in the field is loaded into the corresponding breakpoint register and the breakpoint is enabled.

Breakpoints are described in the section in Chapter 16 titled "Breakpoint-Trace Mode."

## Stop Processor

**Message Type:**  $83_{16}$

**Function:** Stops processor. The processor puts itself into the stopped state. If the processor is in the process-executing state when this IAC is received, the current process is suspended (but not rescheduled).

**Store Processor****Message Type:**86<sub>16</sub>

**Function:** Writes any cached parts of the PRCB (including the processor controls word) back to the PRCB in memory. This IAC allows the PRCB in memory to be updated with any changes that have been made to the fields of the PRCB that are cached in the processor. Refer to the section in Chapter 9 titled "Caching PRCB Fields" for information on the fields in the PRCB that are cached.

**Store System Base****Message Type:** 80<sub>16</sub>

**Parameters:** Fields 1 - 2 Not Used  
Field 3 Destination Physical Address  
Fields 4 - 5 Not Used

**Function:** Stores the current locations of the segment table and the PRCB in a specified location in memory. The base physical address of the segment table is stored in the word starting at the byte specified in field 3, and the base physical address of the PRCB is stored in the next word in memory (field 3 address plus 4).



## Test Pending Interrupts

**Message Type:**

41<sub>16</sub>

**Function:**

Tests for pending interrupts. The processor checks the pending interrupt section of the interrupt table for a pending interrupt with a priority higher than the priority of the current process. If a higher priority interrupt is found, it is serviced immediately. Otherwise, no action is taken.

**Warmstart Processor**

<b>Message Type:</b>	8E <sub>16</sub>	
<b>Parameters:</b>	Fields 1 - 2	Not Used
	Field 3	Physical Address of Segment Table
	Field 4	Physical Address of PRCB
	Fields 4 - 5	Not Used
<b>Function:</b>	<p>Writes any part of the PRCB that has been cached on the chip to the current PRCB in memory, then reestablishes the processor state. This IAC performs a similar function to the restart processor IAC, except that it writes the cached parts of the PRCB to memory before restarting the processor.</p> <p>In restarting itself, the processor first locates the segment table and PRCB from the base physical addresses given in fields 3 and 4. Field 4 may point to the current PRCB or a new PRCB. (Field 3 is only used to locate the eighth segment-table entry, which is used thereafter to locate the segment table.)</p> <p>Next, the processor checks the state field in the processor-controls word in the PRCB and enters that state. If the PRCB state is process-executing, the processor performs a bind action on the process whose SS is in the current-process-SS field in the PRCB.</p> <p>Refer to the section in Chapter 9 titled "Caching PRCB Fields" for information on the fields in the PRCB that are cached.</p>	







Table 12-1 Fault Subtypes

## CHAPTER 12 FAULT HANDLING

This chapter describes the fault handling facilities of the 80960MC processor. The subjects covered include the fault-handling data structures, the required software support required for fault handling, and the fault handling mechanism. A reference section that contains detailed information on each fault type is provided at the end of the chapter.

### OVERVIEW OF THE FAULT-HANDLING FACILITIES

The processor is able to detect various conditions in code or in its internal state (called "fault conditions") that could cause the processor to deliver incorrect or inappropriate results or that could cause it to head down an undesirable control path. For example, the processor recognizes divide-by-zero and overflow conditions on both integer and real-number calculations. It also detects inappropriate operand values, references to incomplete or non-existent architecture-defined data structures, or references to virtual-memory pages that are not currently in physical memory.

The processor can detect a fault while it is working on a process, an interrupt handler, or a fault handler, or while it is in the idle state. (In this chapter, when a process is referred to, it generally also means any interrupt handler or fault handler that may have been invoked while the processor was working on the process.)

When the processor detects a fault, it handles the fault immediately and independently of the process or handler it is currently working on, using a mechanism similar to that used to service interrupts.

A fault is generally handled with a fault-handling procedure (called a fault handler), which the processor invokes through an implicit procedure call. Prior to making the call, the processor saves the state of the current process and in some cases the state of an incomplete instruction. It also saves information about the fault, which the fault handler can use to correct or recover from the condition that caused the fault.

If the fault handler is able to recover from the fault, the processor can then restore the process to its state prior to the fault and resume work on the process. If, on the other hand, the fault has catastrophic effects on the system, facilities are provided that allow the processor to shut itself or the whole system down gracefully.

### FAULT TYPES

All of the faults that the processor detects are predefined. These faults are divided into types and subtypes, each of which is given a number. Table 12-1 lists the faults, arranged by type and subtype.

Fault Type		Fault Record	
No.	Name	No./Bit Position	Name
E	Event	1	Event Notice
D	Descriptor	1	Invalid Descriptor
A	Type	1	Type Mismatch
9	Structural	3	IAC
7	Protection	1	Segment Length
6	Virtual	1	Invalid Segment-Table Entry
5		2	Invalid SS
4	Floating Point	2	Floating Reserved-Encoding
3		4	Floating Incomplete
2		5	Bit 5
1		6	Bit 6
		7	Bit 7
		8	Bit 8
		9	Bit 9
		10	Bit 10
		11	Bit 11
		12	Bit 12
		13	Bit 13
		14	Bit 14
		15	Bit 15



**Table 12-1: Fault Types and Subtypes**

Fault Type		Fault Subtype		Fault Record
No.	Name	No./Bit Position	Name	
1	Trace	Bit 1	Instruction Trace	0xXX01 XX02
		Bit 2	Branch Trace	0xXX01 XX04
		Bit 3	Call Trace	0xXX01 XX08
		Bit 4	Return Trace	0xXX01 XX10
		Bit 5	Prereturn Trace	0xXX01 XX20
		Bit 6	Supervisor Trace	0xXX01 XX40
		Bit 7	Breakpoint Trace	0xXX01 XX80
2	Operation	1	Invalid Opcode	0xXX02 XX01
		4	Invalid Operand	0xXX02 XX04
3	Arithmetic	1	Integer Overflow	0xXX03 XX01
		2	Arithmetic Zero-Divide	0xXX03 XX02
4	Floating Point	Bit 0	Floating Overflow	0xXX04 XX01
		Bit 1	Floating Underflow	0xXX04 XX02
		Bit 2	Floating Invalid-Operation	0xXX04 XX04
		Bit 3	Floating Zero-Divide	0xXX04 XX08
		Bit 4	Floating Inexact	0xXX04 XX10
		Bit 5	Floating Reserved-Encoding	0xXX04 XX20
5	Constraint	1	Constraint Range	0xXX05 XX01
		2	Invalid SS	0xXX05 XX02
6	Virtual Memory	1	Invalid Segment-Table Entry	0xXX06 XX01
		2	Invalid Page-Table-Directory-Entry (PTDE)	0xXX06 XX02
		3	Invalid Page-Table-Entry (PTE)	0xXX06 XX03
7	Protection	Bit 1	Segment Length	0xXX07 XX01
		Bit 2	Page Rights	0xXX07 XX02
8	Machine	1	Bad Access	0xXX08 XX01
9	Structural	1	Control	0xXX09 XX01
		2	Dispatch	0xXX09 XX02
		3	IAC	0xXX09 XX03
A	Type	1	Type Mismatch	0xXX0A XX01
		2	Contents	0xXX0A XX02
C	Process	1	Time Slice	0xXX0C XX01
D	Descriptor	1	Invalid Descriptor	0xXX0D XX01
E	Event	1	Event Notice	0xXX0E XX01

When the processor detects a fault, it records the fault type and subtype in a fault record. It then uses the type number to select a fault handler. The fault handler has the option of using the subtype number to select a specific fault-handling procedure. The fifth column of Table 12-1 shows each fault as it appears in the fault record (the word at offset 40 of the fault record is shown later in this chapter).

For convenience, individual faults are referred to in this manual by their fault-subtype name. Thus a *machine bad-access fault* is referred to as simply a *bad-access fault*, or a *virtual-memory, invalid page-table-directory-entry fault* is referred to as an *invalid PTDE fault*.

For some fault types, multiple subtypes can occur at the same time. Here, each subtype is assigned a separate bit position in the subtype field in the fault record. The fault handler can then use this information to select a specific fault handling scheme to take care of the whole group of fault subtypes.

## FAULT-HANDLING METHODS

The processor handles faults using one or more of the following methods:

- Implicit procedure call to a fault handler
- Implicit procedure call to an override fault handler
- System-error interrupt that invokes a special interrupt handler through the interrupt mechanism
- Change of the processor state to stopped

These four fault-handling methods provide the processor with an efficient mechanism for recovering from faults or for gradually degrading its processing activity when serious or catastrophic fault conditions are encountered. The scenario for handling faults with this mechanism is as follows.

### Normal Fault-Handling Method

When a fault occurs while the processor is executing a process, the processor determines the fault type, then selects a fault handler for that type from an architecture-defined data structure called the *fault table*. It then invokes the fault handler (by means of an implicit call). As described later in this chapter, the fault-handler call can be a local call (call-extended operation), a local procedure-table call (local system-call operation), a supervisor call, or a trace-fault-handler-procedure-table call.

Before the processor begins executing the fault-handler procedure, it creates a fault record on its current stack (i.e., the stack being used by the fault handler). This record includes information on the state of the process and data on the fault. If the fault occurred while the processor was in the midst of executing an instruction, a resumption record for the instruction may also be saved on the stack.

Following the creation of the fault and resumption records, the processor begins executing the selected fault-handler procedure.

This same procedure call method is used to handle faults that occur while the processor is servicing an interrupt or that occur while the processor is working on another fault handler.

### Overrides

If a fault should occur while the processor is selecting a fault handler (i.e., between the time the processor begins storing the fault and resumption records for a fault and the time it begins work on the fault handler for that fault), an override is said to occur. When an override occurs, the processor stores a fault record for both faults (i.e., the primary fault and the secondary fault). The processor then invokes an override fault handler to perform the recovery action.

The action of the override-fault handler is software dependent. Commonly, the override-fault handler handles the secondary fault, then returns. On the return, the processor refaults on the primary fault (that is, recreates the primary fault). That fault is then handled as described in the previous section.

A common cause of an override condition is a virtual-memory fault that occurs while the processor is trying to store the fault record or create a stack frame for the fault handler. For example, assume that the execution of a divide instruction results in an arithmetic-zero-divide fault being generated, and that, while storing the fault record for this fault, a virtual-memory fault is generated. Here, the processor saves the fault data on both faults (the primary arithmetic-zero-divide fault and the secondary virtual-memory fault). The override-fault handler then handles the virtual-memory fault, by copying the required page into memory. On the return from the override-fault handler, the processor refaults on the arithmetic-zero-divide fault, which is handled by the arithmetic-fault handler.

### System-Error Interrupt

If a second override should occur (i.e., if a fault occurs between the time the processor begins storing the fault record for an override fault and the time it begins work on the fault handler for the override fault), the processor handles the second override by means of a system-error interrupt.

Here, the processor saves the process state and fault information for all three faults in the PRCB, then performs a recovery action, using a interrupt handler that it accesses through the interrupt table. (Interrupt vector 248 in the interrupt table is reserved for system-error interrupts.) The processor does not provide a mechanism for returning from a system-error interrupt handler. A system-error interrupt thus represents a fatal condition, which results at the very least in the current process being aborted.

This system-error interrupt mechanism is also used when a fault occurs while the processor is in the idle or stopped state. For example, assume that the processor has suspended one process and is attempting to dispatch another process. While the processor is in between processes, it is in the idle state. If a structural fault occurs while the processor is attempting to dispatch a process, this fault results in a system-error interrupt.

## Halt

Finally, if a fault occurs while the processor is generating a system-error interrupt, the processor halts. As part of the halt action, the processor collects as much information as possible about the last fault, then puts itself into the stopped state.

## Multiple Fault Conditions

It is possible for multiple fault conditions to occur simultaneously. For certain fault types, such as trace faults or protection faults, bit positions in the fault-subtype field are used to indicate the occurrence of multiple faults of the same type. As a general rule, however, the processor does not indicate situations where multiple faults occur. Instead, it generates one of the faults and does not report on the faults that were not generated.

## SOFTWARE REQUIREMENTS FOR HANDLING FAULTS

To use the processor's fault-handling facilities, the following data structures and procedures must be present in memory:

- Fault table
- Trace-Fault-Handler Procedure Table
- Fault-Handler Procedure Table (Optional)
- Fault-Handler Procedures
- Interrupt Table
- Interrupt Stack

Software should generally load these items in memory as part of the initialization procedure. Once they are present in memory and pointers to them have been entered in the appropriate data structures, the processor then handles faults automatically and independently from software.

### NOTE

If the virtual-memory-management features of the processor are being used, the fault-handling data structures should be frozen in memory (i.e., they should never be swapped out of memory).

Requirements for the fault table, trace-fault-handler-procedure table, and fault-handler procedures are given in the following sections. Requirements for the interrupt table and interrupt stack are given in Chapter 10.

## FAULT TABLE

The fault table provides the processor with a pathway to the fault-handler procedures. As shown in Figure 12-1, there is one entry in the fault table for each fault type plus an entry for overrides. When a fault occurs, the processor uses the fault type to select an entry in the fault table. From this entry, the processor then obtains a pointer to the fault-handler procedure for the type of fault that occurred.

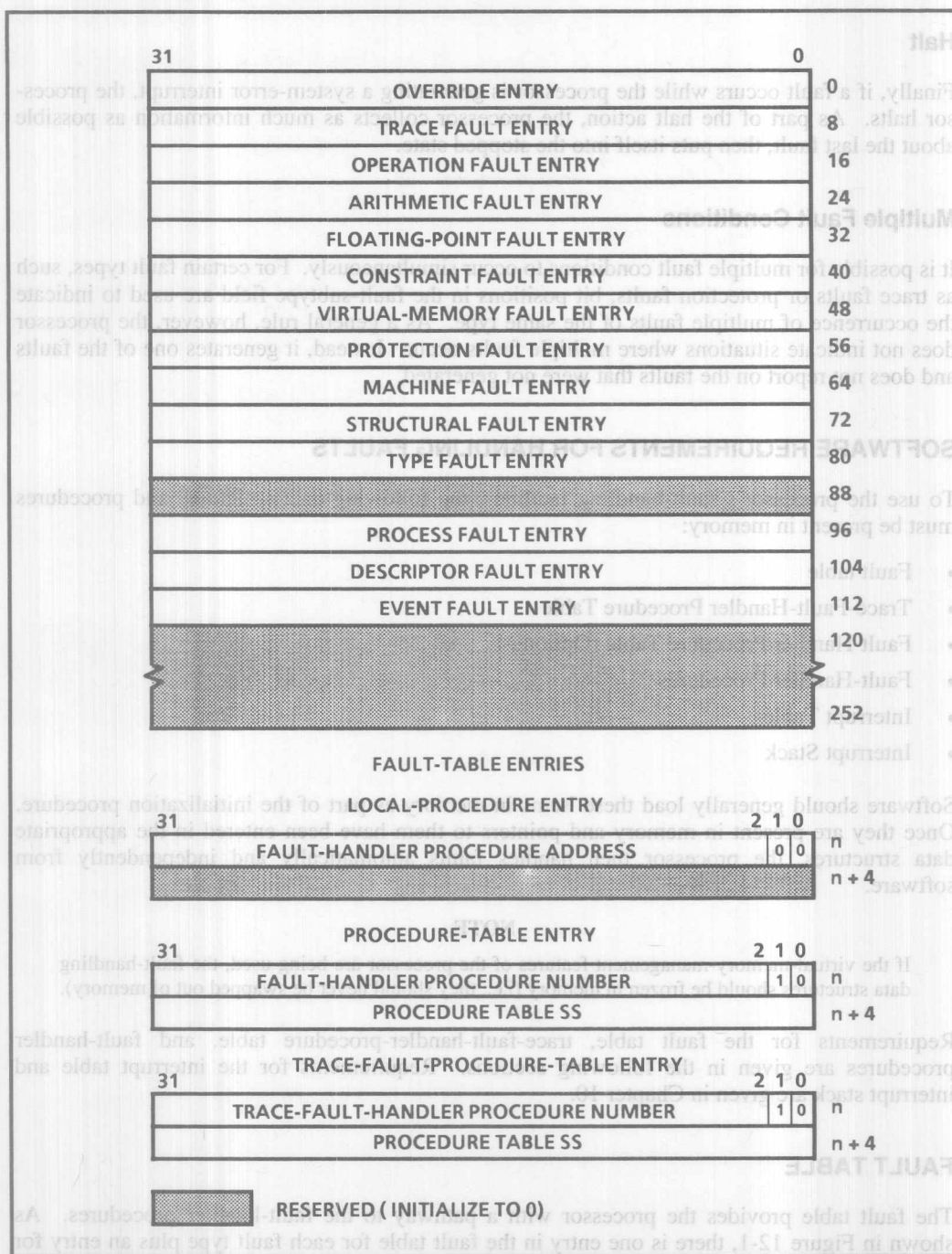


Figure 12-1: Fault Table and Fault-Table Entries



Once a fault-handler procedure has been called, it has the option of reading the fault subtype or subtypes from the fault record to determine the appropriate fault recovery action.

### Location of the Fault Table in Memory

The fault table is located in physical memory. The processor obtains a physical-address pointer to the fault table from the PRCB.

The fault table is placed in physical memory for two reasons: to avoid a virtual memory fault while handling a fault and to provide access to the fault-handling procedures during initialization.

### Fault-Table Entries

As shown at the bottom of Figure 12-1, three types of fault-table entries are allowed: a local-procedure entry, a procedure-table entry, and a trace-fault-handler-procedure-table entry. Each entry type is two words long. The entry-type field (bits 0 and 1 of the first word of the entry) and the SS in the second word of the entry determines the entry type.

A local-procedure entry (entry type  $00_2$ ) provides an instruction pointer (address in the address space) for the fault-handler procedure. Using this entry, the processor invokes the specified fault handler by means of an implicit call-extended operation (similar to that performed for the **callx** instruction). The second word of a local-procedure entry is reserved. It should be set to zero when the fault table is created and not accessed after that.

A procedure-table entry provides a procedure number in a procedure table. This entry must have an entry type of  $10_2$  and an SS for the procedure table in the second word. Using this entry, the processor invokes the specified fault handler by means of an implicit call-system operation (similar to that performed for the **calls** instruction). Fault-handling procedures in the procedure table can be local procedures or supervisor procedures.

The procedure table can be the system procedure table that the kernel provides as an entry point for supervisor calls or a special procedure table, which is reserved for fault-handling procedures alone. If a special, fault-handler procedure table is created, it must have the same format as the procedure table shown in Figure 4-4. The supervisor stack pointer in this table should point to the same stack that is pointed to in the system procedure table.

The trace-fault-handler-procedure-table entry provides a procedure number in a special procedure table called the *trace-fault-handler procedure table*. This entry must have an entry type of  $10_2$  and an SS for the trace-fault-handler procedure table in the second word. The function of this entry is described in the following section titled "Handling Trace Faults."

To summarize, a fault handler can be invoked through the fault table in any of four ways: a local procedure call; a local procedure-table call; a supervisor call; or a trace-fault-handler procedure-table call.



## TRACE-FAULT HANDLING

When handling trace faults, the 80960 architecture requires that tracing be disabled (i.e., the trace-enable flag of the process controls must be set to 0). To support this requirement, the architecture defines a special trace-fault-handler procedure table. This procedure table has the same structure as the procedure table shown in Figure 4-4, but with the following two restrictions:

- All entries must be supervisor entries (10<sub>2</sub> in bits 0 and 1).
- The trace control flag (byte 12, bit 0) must be set to 0.

The supervisor stack pointer in the trace fault-handler procedure table should be the same as the stack pointer given in the system procedure table.

The effect of these restrictions is that on a call to a trace-fault handler routine, the processor saves the current state of the trace-enable flag and then clears the flag to disable tracing. On the return from the trace fault handler, the processor automatically restores the trace-enable flag to the state it was in prior to the trace fault.

The trace-fault-handler procedure table will generally have only one procedure entry, which points to the trace-fault handler procedure. However, this procedure table can be used as a pathway to other fault-handler routines.

This method of handling trace faults must always be used except for the following circumstances:

- If tracing is never going to be used (i.e., the trace-enable flag of the process controls is always set to 0), the trace-fault-handler procedure table is not required.
- If tracing is never going to be used on supervisor calls, the system-procedure table can be used in place of the trace-fault-handler procedure table, since the trace-control flag of the system-procedure table will then be set to 0.

In the latter case, the trace-fault handler must still be called with a supervisor call.

## FAULT-HANDLER PROCEDURES

The fault-handler procedures are generally located in region 3 of virtual memory, although they can be located in any region. By locating the procedures in region 3, the processor always has access to them whether it is bound to a process or not. (The fault-handler procedures can also be located in physical memory if the physical-addressing translation mode is being used.) Each procedure must begin on a word boundary.

The processor can execute the procedure in the user mode or the supervisor mode, depending on the type of fault table entry. If a fault handler is intended to be executed from the supervisor mode, the page rights for the page or pages that contain the handler may be set for supervisor-only access.

**NOTE**

To resume work on a process at the point where a fault occurred (following the recovery action of the fault handler), the fault handler must be executed in the supervisor mode. The reason for this requirement is described in a following section titled "Returning with Resumption."

**Possible Fault-Handler Actions**

Many of the faults that occur can be recovered from easily. For example, recovery from an invalid PTE fault merely involves copying the page from the disk into memory and marking the page-table entry as valid.

When recovery from the fault is possible, the processor's fault-handling mechanism allows the processor to automatically resume work on the process or interrupt it was working on when the fault occurred. The resumption action is initiated with a **ret** instruction in the fault-handler procedure.

If recovery from the fault is not possible or not desirable, the fault handler can take one of the following actions, depending on the nature and severity of the fault condition (or conditions, in the case of multiple faults):

- Return to a point in the process or interrupt code other than the point of the fault
- Suspend the current process and rebind it to the processor
- Suspend the current process and bind a new process to the processor
- Suspend the current process and place the processor in the idle or stopped state
- Explicitly write the fault record and instruction resumption record into the fields provided for them in PRCB, suspend the current process, and place the processor in the idle or stopped state.
- Explicitly write the fault record and instruction resumption record into the fields provided for them in PRCB and place the processor in the idle or stopped state, without suspending the current process.
- Place the processor in the idle or stopped state without explicitly saving the process state or the fault information.

When working with the processor at the development level, a common action of the fault handler is to save the fault and processor state information and make a call to a debugging device such as a debugging monitor. This device can then be used to analyze the fault.

**Process and Instruction Resumption Following a Fault**

Faults can occur prior to the execution of the faulting instruction (i.e., the instruction that causes the fault), during the instruction, or after the instruction. When the fault occurs before the faulting instruction is executed, the instruction can theoretically be executed on the return from the fault handler. So, the fault can be handled in such a way as to not interrupt in the control flow of the process.

When a fault occurs during or after the instruction that caused a fault, the fault may be accompanied by a change in the process state such that the execution of the process can not be resumed after the fault has been handled. For example, when an integer-overflow fault occurs, the overflow value is stored in the destination. If the destination register was the same as one of the source registers, the source value is lost, making recovery from the fault impossible.

In general, resumption of process execution with no changes in the process's control flow is always possible with the following fault types or subtypes:

- All Operation Subtypes
- Arithmetic Zero-Divide
- All Floating-Point Subtypes Except Floating Inexact
- All Constraint Subtypes
- All Trace Subtypes
- Invalid Descriptor
- All Virtual Memory Subtypes
- Time Slice
- Event Notice

Resumption of the process may or may not be possible with the following fault types and subtypes:

- Integer Overflow
- Floating Inexact
- All Structural Subtypes
- Bad Access
- All Protection Subtypes
- All Type Subtypes

The effect that specific fault types have on a process is given in the fault reference section at the end of this chapter under the heading "Process State Changes."

### Returning With Resumption

As described above, certain faults do not change the state of the process when they occur, even if the execution of the instruction was suspended as part of the fault-generation mechanism. Here, the processor allows work on a process to be resumed at the point where the fault occurred (including resumption of a suspended instruction), following a return from a fault handler. The resumption mechanism is similar to that provided for returning from an interrupt handler.

To use this mechanism, the fault handler must be invoked using a supervisor call. This method is required because to resume work on the process and a suspended instruction at the point where the fault occurred, the saved process controls in the fault record must be copied back

into the process's PCB on the return from the fault handler. The processor only performs this action if the processor is in the supervisor mode on the return.

If the fault handler is invoked with a local-procedure call or a local-procedure-table call, the return IP determines where in the process the processor resumes work, following a return from a fault handler. Here, the return is handled in a similar manner to a return from an explicit call with a **call** or **callx** instruction.

The return IP (referred to later in this chapter as the saved IP) is saved in the RIP register (r2) of the stack frame that was in use when the fault occurred. This IP may be the instruction the processor faulted on or the next instruction that the processor would have executed if the fault had not occurred. In either case, the resumption record is not used, so the processor might continue work on the process without completing the instruction that the fault occurred on.

A fault handler should thus be invoked with a local-procedure or local-procedure-table call only if it is not required or desirable to resume the process at the point where the fault occurred. The section later in this chapter titled "Returning Without Resumption" discusses returning to a point in the process code other than the point of the fault.

## Return Without Resumption

There may be situations where the fault handler needs to return to a point in the process other than where the fault occurred. This can be done by altering the return IP in the previous frame. However, if resumption information was collected with the fault (resulting in the resume flag being set in the saved process controls), such a return can cause unpredictable results.

To predictably perform a return from a fault handler to an alternate point in the process, the fault handler should perform the following two steps:

1. Flush the local register sets to the stack with a **flushreg** instruction.
2. Clear the following information in the process-controls field of the fault record before the return: the resume and trace-fault-pending flags; the internal state field.

### NOTE

This technique should be used carefully and only in situations where the fault handler is closely coupled with the application program. Also, a return of this type can only be performed if the processor is in supervisor mode prior to the return.

## Aborting a Process

Where it is not possible to return to the process in which a fault occurred, the fault handler can be designed to abort the process. Several possible actions that a fault handler can take when aborting a process are given in the section earlier in this chapter titled "Possible Fault-Handler Actions."

## FAULT CONTROLS

Certain fault types and subtypes have masks or flags associated with them that determine whether or not a fault is generated when a fault condition occurs. Table 12-2 lists these flags and masks, the data structures in which they are located, and the fault subtype they affect.

**Table 12-2: Fault Flags or Masks**

Flag or Mask Name	Location	Fault Affected
Integer Overflow Mask	Arithmetic Controls	Integer Overflow
Floating Overflow Mask	Arithmetic Controls	Floating Overflow
Floating Underflow Mask	Arithmetic Controls	Floating Underflow
Floating Invalid Operation Mask	Arithmetic Controls	Floating Invalid Operation
Floating Zero-Divide Mask	Arithmetic Controls	Floating Zero-Divide
Floating-point Inexact Mask	Arithmetic Controls	Floating Inexact
No Imprecise Faults Flag	Arithmetic Controls	All Imprecise Faults
Refault Flag	Process Controls	All Faults
Trace-Enable Flag	Process Controls	All Trace Faults
Trace-Mode Flags	Trace Controls	All Trace Faults
Event-Fault Request Flags	PCB	Event Notice Fault

The integer and floating-point mask bits inhibit faults from being raised for specific fault conditions (i.e., integer overflow and floating-point overflow, underflow, zero divide, invalid operation, and inexact). The use of these masks is discussed in the fault-reference section at the end of this chapter. Also, the floating-point fault masks are described in Chapter 7 in the section titled "Exceptions and Fault Handling."

The no-imprecise-faults (NIF) flag controls the synchronizing of faults for a category of faults called imprecise faults. This flag should be set to 1. The function of this flag is described later in this chapter in the section titled "Precise and Imprecise Faults."

The refault flag causes a fault to be generated on a return from a fault handler. This flag is used in the handling of override conditions and can also be used by the kernel. Refer to the sections in this chapter titled "Generating Faults" and "Override Fault-Handling Action" for further information on the refault flag.

The trace-mode flags (in the trace controls) and trace-enable flag (in the process controls) support trace faults. The trace-mode flags enable trace modes; the trace-enable flag enables the generation of trace faults. The use of these flags is described in the fault reference section on trace faults at the end of this chapter. Further discussion of these flags is provided in Chapter 16 in the section titled "Trace-Enable and Trace-Fault-Pending Flags."

The event-fault request flags cause an event-notice fault to be generated under specific circumstances. These flags are discussed in the fault reference section on event faults at the end of this chapter.



## FAULTS AND INTERRUPTS

If an interrupt occurs during an instruction that will fault, that has just faulted, or that has faulted while the processor is in the midst of selecting the fault handler, the processor will handle the fault in the following way. It completes the selection of the fault handler, then services the interrupt just prior to executing the first instruction of the fault handler. On returning from the interrupt, the fault is handled.

## PROCESSING TIMING WHILE HANDLING A FAULT

When a fault occurs while the processor is executing a process, the processor continues to count process time (i.e., count down the residual-time-slice value) while it is executing the fault-handler procedure. If an end-of-time-slice event occurs while the fault handler is being executed, the processor handles the event just as it would if the event occurred while the process was being executed. For example, if the process-timing controls are configured to suspend a process at the end of a time slice, the processor suspends the process in the midst of the fault-handler routine. The next time the process is dispatched, the processor begins working on the fault handler where it left off.

## GENERATING A FAULT

The processor generates faults implicitly when fault conditions occur and explicitly at the request of software. Most faults are generated implicitly. The fault control bits described in the previous section allow the implicit generation of some faults to be either enabled (as with the trace faults) or masked (as with the floating-point faults).

The following paragraphs describe faults that software can cause to be generated explicitly.

### Fault-If and Mark Instructions

Two sets of instructions allows faults to be generated explicitly anywhere within an application program, kernel procedure, interrupt handler, or fault handler. The fault-if instructions (**faulte**, **faultne**, **faultl**, **faultle**, **faultg**, **faultge**, **faulto**, and **faultno**) allow a conditional fault to be generated. When one of these instructions is executed, the processor checks the condition code bits in the arithmetic controls, then generates a constraint-range fault if the condition specified with the instruction is met.

The **mark** and force mark (**fmark**) instructions allows a breakpoint trace fault to be generated anywhere in the instruction stream.

### Event-Notice Fault

The process-notice field in the PCB (shown in Figure 13-3) has two event-fault request flags. When these flags are set, an event notice fault is generated in either of the following two instances:



- While the process associated with the PCB is being bound to the processor.
- If the process is already bound to the processor and the process notice IAC is sent to the processor.

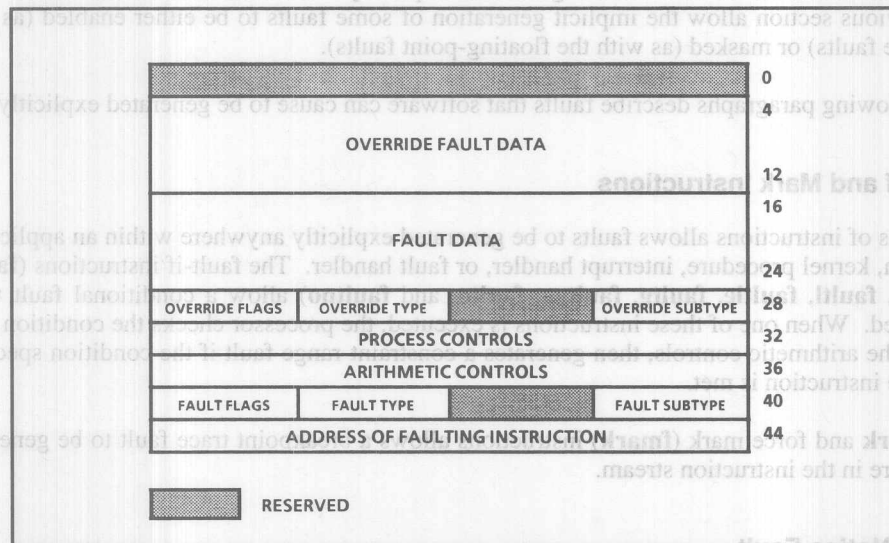
In the latter case, software would set the event-fault request flags after the process had been bound to the processor, then send the IAC.

This faulting technique is used primarily by kernel procedures within multiprocessor systems. It can only be used within a procedure that is being executed in supervisor mode.

Further information on the event-notice fault is given in the fault reference section at the end of this chapter.

## FAULT RECORD

When a fault occurs, the processor records information about the fault in a fault record. (The location of the fault record is described later in this chapter in the section titled "Location of the Fault and Resumption Records.") The fault handler and processor use the information in the fault record to recover from or correct the fault condition and resume execution of the process. Figure 12-2 shows the structure of the fault record. The use of the fields in this record are described in the following paragraphs.



**Figure 12-2: Fault Record**

The fault record provides space for fault information on two faults: a normal fault and (if one occurs) an override fault.

The type number (byte ordinal) of a fault is stored in the fault type (normal fault) and override type (override fault) fields; the subtype number or bit positions (byte ordinal) is stored in the fault subtype (normal fault) and override subtype (override fault) fields.

Two sets of eight flags, fault flags field (normal fault) and override flags field (override fault) are also provided. Of these flags, only F0 and F1 (bits 24 and 25) are used. Most of the faults do not use these flags, in which case the flags have no defined values.

The address-of-the-faulting-instruction field contains the IP of the instruction that caused the fault or that was being executed when the fault occurred.

The states of the process controls and arithmetic controls at the time that a normal fault is generated are stored in their respective fields in the fault record. This information is used to resume work on the process after the fault has been handled.

Finally, a three-word fault data field is provided for both a normal fault and an override fault. The information that is stored in these fields depends on the type of fault that occurs. Any part of a fault-data field that is not used for a particular fault has no defined value. The information that is stored in these fields for each fault type is given in the fault reference section at the end of this chapter.

### Saved Instruction Pointer

The saved IP (the RIP that is saved in r2 of the stack frame in use when the fault occurred) is also part of the fault information that the processor saves when a fault occurs. This IP generally points to the next instruction that the processor would have executed if the fault had not occurred, although it may point to the faulting instruction. It is this instruction that the processor begins working on when the return from the fault handler is initiated.

### Resumption Record

If the processor suspends an instruction as the result of a fault, it creates a 48-byte resumption record. The criteria that the processor uses to determine whether or not to suspend an instruction and the structure of the resumption record are the same as are used when an interrupt occurs.

### Location of the Fault and Resumption Records

The fault and instruction-resumption records are stored in the fault handler's stack, the PRCB, or both places, depending on the circumstances under which the fault occurred. If the fault occurs while the processor is doing any of the following things, the fault and resumption records is stored in the stack that the processor will use to execute the fault-handler procedure:

- Executing a process
- Servicing an interrupt
- Handling another fault

• Selecting a fault handler (first override fault)

As shown in Figure 12-3, this stack can be the local stack, the supervisor stack, or the interrupt stack. The fault record begins at the byte address of the new frame pointer (NFP) minus 48, and the instruction resumption record begins at NFP minus 96.

If the fault occurs while the processor is doing any of the following things, the fault record is stored in the PRCB:

- Selecting the override-fault handler (second override fault)
- In the idle processor-state

Both of the above situations cause a system-error interrupt. When the system-error interrupt is the result of a second override fault, the fault-record is stored in two fields in the PRCB: the system-error-fault field (bytes 72 through 75) and the system-error-fault-record field (bytes 128 through 175).

The fault record for the first two faults (the normal fault and the first override fault) is stored in the system-error-fault record in the format shown in Figure 12-2. The fault type and subtype of the second override fault is stored in the system-error-fault field, but no fault data is stored for this fault.

The system-error interrupt handler thus has the following information available to it for the purposes of handling a system-error interrupt: (1) the process state when the first fault occurred, (2) complete fault data on the first two faults, and (3) the fault type and subtype of the third fault.

When the system-error interrupt occurs while the processor is in the idle state, a record for this fault is stored in the system-error-fault-record field. Here, the system-error-fault field is not used, because the fault type and subtype are contained in the system-error-fault-record field.

Finally, if a fault occurs while the processor is selecting the system-error fault handler (which causes a halt), the fault information collected in the PRCB for all the faults that occurred up through the first system-error interrupt is maintained. However, no fault information on the fault that occurred while the system-error interrupt handler was being selected is recorded before the processor places itself in the stopped state.

## FAULT-HANDLING ACTION

Once a fault has occurred, the processor saves the process state, calls the fault handler, and restores the process state (if this is possible) once the fault recovery action has been completed. No software other than the fault-handler procedures is required to support this activity.

The following sections describe the action that the processor takes while handling a fault.

- Executing a process
- Servicing an interrupt
- Handling another fault

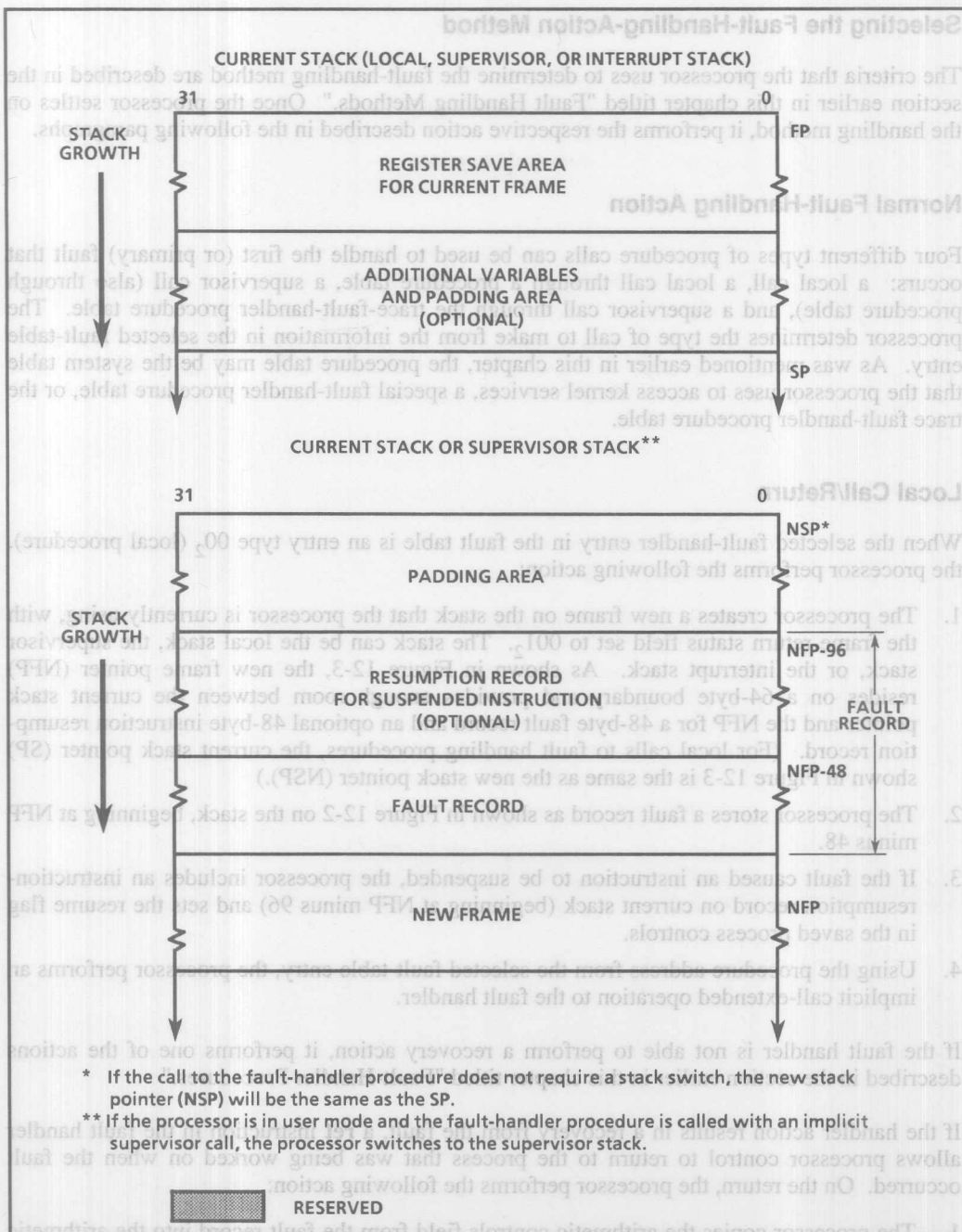


Figure 12-3: Storage of the Fault and Resumption Records on the Stack

## Selecting the Fault-Handling-Action Method

The criteria that the processor uses to determine the fault-handling method are described in the section earlier in this chapter titled "Fault Handling Methods." Once the processor settles on the handling method, it performs the respective action described in the following paragraphs.

### Normal Fault-Handling Action

Four different types of procedure calls can be used to handle the first (or primary) fault that occurs: a local call, a local call through a procedure table, a supervisor call (also through procedure table), and a supervisor call through the trace-fault-handler procedure table. The processor determines the type of call to make from the information in the selected fault-table entry. As was mentioned earlier in this chapter, the procedure table may be the system table that the processor uses to access kernel services, a special fault-handler procedure table, or the trace fault-handler procedure table.

### Local Call/Return

When the selected fault-handler entry in the fault table is an entry type  $00_2$  (local procedure), the processor performs the following action:

1. The processor creates a new frame on the stack that the processor is currently using, with the frame-return status field set to  $001_2$ . The stack can be the local stack, the supervisor stack, or the interrupt stack. As shown in Figure 12-3, the new frame pointer (NFP) resides on a 64-byte boundary and provides enough room between the current stack pointer and the NFP for a 48-byte fault record and an optional 48-byte instruction resumption record. (For local calls to fault handling procedures, the current stack pointer (SP) shown in Figure 12-3 is the same as the new stack pointer (NSP).)
2. The processor stores a fault record as shown in Figure 12-2 on the stack, beginning at NFP minus 48.
3. If the fault caused an instruction to be suspended, the processor includes an instruction-resumption record on current stack (beginning at NFP minus 96) and sets the resume flag in the saved process controls.
4. Using the procedure address from the selected fault-table entry, the processor performs an implicit call-extended operation to the fault handler.

If the fault handler is not able to perform a recovery action, it performs one of the actions described in the section earlier in this chapter titled "Fault-Handler Procedures."

If the handler action results in a recovery from the fault, a **ret** instruction in the fault handler allows processor control to return to the process that was being worked on when the fault occurred. On the return, the processor performs the following action:

1. The processor copies the arithmetic controls field from the fault record into the arithmetic controls register in the processor.
2. If the resume flag of the process controls is set, the processor reads the resumption record from the stack.



3. The processor deallocates the stack frame created for the fault handler.
4. The processor then resumes work on the process it was working on when the fault occurred at the instruction in the return IP register.

#### NOTE

The saved process controls are not copied back into the PCB, unless the execution mode is supervisor at the time of the return, which would not ordinarily be the case with a local call to the fault handler. Thus any changes in the process controls that the fault handler makes become part of the process state when the processor resumes work on the process.

### Local Procedure-Table Call/Return

When the fault-handler entry selects an entry in a special fault-handler procedure table (or the system procedure table) and the procedure-table entry is for a local procedure, the processor performs the same action as is described in the previous section for a local-procedure call and return. The only difference is that the processor gets the address of the fault handler from the procedure table rather than from the fault table.

### Supervisor Call/Return

When the fault-handler entry selects an entry in a fault-handler procedure table (or the system procedure table) and the procedure-table entry is for a supervisor procedure, the processor performs the following actions:

1. If the processor is in user mode when the fault occurs, the processor then reads the supervisor-stack pointer from the procedure table and switches to the supervisor stack. The supervisor-stack pointer then becomes the NSP shown in Figure 12-3. Also, the execution mode is set to supervisor.
2. If the processor is already in supervisor mode when the fault occurs, the processor stays on the current stack. Here, the SP and the NSP in Figure 12-3 are the same. (If the processor was executing a supervisor procedure when the fault occurred, the current stack will be the supervisor stack; if it was executing an interrupt-handler procedure, the current stack will be the interrupt stack. The processor switches to supervisor mode when handling interrupts.)
3. The processor copies the state of the trace-control flag (byte 12, bit 1) of the procedure table into the trace-enable flag field of the process controls.
4. The processor creates a new frame on the current stack (as described above for the local call); stores the fault record and optional instruction resumption record in the areas allocated for them on the stack; and begins work on the fault-handler procedure selected from the procedure table.

On a return from the fault handler, the processor performs the following actions:

1. The processor copies the arithmetic-controls field from the fault record into the arithmetic-controls register in the processor.
2. If the processor is in supervisor mode prior to the return from the fault handler (which it should be), it copies the saved process controls into its internal process controls.



3. If the resume flag of the process controls is set, the processor reads the resumption record from the stack.
4. The processor deallocates the stack frame created for the fault handler and returns to the stack it was using prior to the call to the fault handler routine.
5. If the processor was in user mode prior to the supervisor call, the mode is set back to user mode; otherwise, the processor remains in supervisor mode.
6. The processor resumes work on the process it was working on when the fault occurred, at the instruction in the return IP register.

The restoration of the process controls causes any changes in the process controls through the action of the fault handler to be lost. In particular, if the **ret** instruction from the fault handler caused the trace-fault-pending flag in the process controls to be set, this setting would be lost on the return.

### Trace-Fault-Handler Call/Return

When the fault table entry is for a trace fault, the processor performs the same action as is described in the previous section for a supervisor call and return. The only difference is that the processor uses the trace-fault-handler procedure table instead of the normal-fault-handler procedure table (or system procedure table).

### Override Fault-Handling Action

When an override fault occurs, the processor can call the override-fault handler using any of the techniques described above (local call, local call through a procedure table, supervisor call, or trace-fault-handler procedure table call). The processor performs the same actions on the call and return as described above except for the follow things.

When calling the override-fault handler, the processor performs the following additional actions:

1. The processor saves an override fault record (that contains the primary and the secondary fault data) on the stack.
2. The processor sets the refault and resume flags in the saved process controls. (The resume flag is set even if a resumption record is not saved.)
3. The processor begins work on the selected override-fault handler.

The override-fault handler can be designed to attempt to correct both faults or correct the override fault and then refault on the original fault, allowing the fault handler for that fault to be called. The latter technique is allowed only if the override-fault handler is called with a supervisor procedure call.

On the return from the override-fault handler, the processor performs the following additional actions:

1. If the processor is in user mode on the return from the fault handler, the saved arithmetic controls are copied into the arithmetic controls register and the processor begins work at the point in the process or interrupt designated with the saved IP.

2. If the processor is in supervisor mode on the return from the fault handler, the saved arithmetic controls are copied into the arithmetic controls register and the saved process controls are copied into the PCB for the process being resumed. The refault and resume flags are then cleared, and the processor refaults on the original (first) fault.

#### NOTE

If the fault handler is not called with a supervisor call, the override-fault handler must handle both the original fault and the override fault. If this is not done, the process might be put into an unpredictable state on the return from the fault handler.

### System-Error-Interrupt Action

When a system-error interrupt occurs, the processor collects data on the faults that caused the condition and calls the system-error interrupt fault handler. The processor does not, however, provide a mechanism for resuming the process, once the handling of the interrupt is complete.

When a system-error interrupt occurs as the result of a second override fault, the processor takes the following action:

1. The processor stores the fault record for the original fault and the first override fault in the system-error-fault-record field of the PRCB.
2. The processor stores the type and subtype of the second override fault in the system-error fault field of the PRCB.
3. The processor switches to the interrupt stack.
4. The processor performs an implicit call operation to vector 248 (the predefined system-error interrupt vector) in the interrupt table.

When a system-error interrupt occurs as the result of a fault occurring while the processor is in the idle state, the processor takes the following action:

1. The processor stores the fault record for the fault in the system-error-fault-record field of the PRCB.
2. The processor switches to the interrupt stack.
3. The processor performs an implicit call operation to vector 248 (the predefined system-error interrupt vector) in the interrupt table.

When a system-error interrupt occurs, the processor does not provide a mechanism for resuming processing at the point that the fault occurred as with the other fault conditions described in this chapter. The action of the system-error interrupt fault-handler is limited to the following actions:

- Suspend the current process and rebind it to the processor.
- Rebind the current process to the processor, without suspending the process first.
- Suspend the current process and bind the processor to a new process.
- Bind the processor to a new process, without suspending the current process first.

- Suspend the current process and place the processor in the idle or stopped state.
- Place the processor in the idle or stopped state, without suspending the current process.
- Call a system debugging monitor for use in analyzing the fault data.
- Write the fault record and other pertinent state data to disk memory, then shut down the system, with or without first suspending the current process.
- Shut down the system without explicitly saving state or fault information.
- In a multiprocessor system, a second processor might be called upon to reinitialize the processor with a restart processor IAC.

### Halt Action

When a fault occurs while the processor is selecting the system-error interrupt handler, the processor takes the following action:

1. If possible, it stores a fault record for the latest fault in the PRCB. This is only possible if the system-error interrupt occurred while the processor was in the idle state.
2. It places itself in the stopped state and asserts the #FAILURE pin.

When the processor experiences enough faults to halt it in the manner described above, its resulting state often prohibits it from reliably executing instructions or even reliably accessing memory. A reinitialization of the processor either through a restart processor IAC or a hardware reset is generally required. If the system uses multiple processors, the still active processor can attempt to save the fault record for later use in a diagnostics routine, before the stopped processor is restarted.

### PRECISE AND IMPRECISE FAULTS

As described in the section in Chapter 3 titled "Register Scoreboarding," the 80960MC processor is, in some instances, able to execute instructions concurrently. When two instructions are being executed concurrently, it is possible for them to generate faults simultaneously. When this occurs, one of the faults may not be generated or may be generated out of order, making it impossible to recover from that fault.

The processor provides two mechanisms to allow the circumstances under which faults are generated to be controlled. These mechanisms are the no imprecise faults flag (NIF) in the arithmetic controls and the synchronize faults instruction (**syncf**). The following paragraphs describe how these mechanisms can be used.

Faults are grouped into the following categories: precise, imprecise, and asynchronous. Precise faults are those that are intended to be recoverable by software. For any instruction that can generate a precise fault, the processor will (1) not execute the instruction if an unfinished prior instruction will fault and (2) not execute subsequent out-of-order instructions that will fault.

The following faults are always precise:

- trace
- virtual memory
- protection
- descriptor faults

Imprecise faults are those that in some instances are allowed to occur and not be generated or be generated out of order. These faults include the following:

- operation
- arithmetic
- floating point
- constraint
- structural
- type

Asynchronous faults are those whose occurrence has no direct relationship to the instruction pointer. This category includes the machine, event, and process faults.

The NIF controls whether or not imprecise faults are allowed. When this flag is set, all faults must be precise. In this mode, the ability to execute instructions concurrently is essentially disabled. All faults that occur are generated.

When the NIF is clear, faults in the imprecise category can in some instances occur and not be generated. In this mode, the following conditions hold true:

1. When an imprecise fault occurs, the saved IP is undefined (but the address of the faulting instruction in the fault record is valid).
2. If instructions are executed concurrently when an imprecise fault occurs, the results produced by these instructions are undefined.
3. If instructions are executed out-of-order and multiple imprecise faults occur, only one of the faults is generated. The one that is selected is not predictable.

The **syncf** instruction forces the processor to complete execution of all instructions that occur prior to the **syncf** instruction and to generate all faults, before it begins work on instructions that occur after the **syncf** instruction. This instruction has two uses. One use is to force faults to be precise when the NIF is clear. The other use is to insure that all instructions are complete and all faults generated in one block of code before execution of another block of code (for example, on Ada block boundaries when the blocks have different exception handlers).

The intent of these fault-generating modes is that compiled code should execute with the NIF clear, using the **syncf** instruction where necessary to ensure that faults occur in order. In this mode, imprecise faults are considered as catastrophic errors from which recovery is not needed.

If recovery from one or more of the imprecise faults is required (for example, a program that needs to handle unmasked floating-point exceptions and recover from them) and the fault handler cannot be closely coupled with the application to perform recovery even if the faults are imprecise, the NIF should be set. Executing with the NIF set will likely lead to slower execution times.

## FAULT REFERENCE

This section describes each of the fault types and subtypes and gives detailed information about what is stored in the various fields of the fault record. The section is organized alphabetically by fault type.

### Fault-Reference Notation

The following paragraphs describe the information that is provided for each fault type.

### Fault Type and Subtype

The fault-type section gives the number entered in the fault-type field of the fault record for the given fault type. The fault-subtype section lists the fault subtypes and their associated number or bit position in the fault-subtype field of the fault record.

### Function

The function section gives a general description of the purpose of the fault type, then describes the purpose of each of the fault subtypes in detail. It also describes how the processor handles each fault subtype.

### Fault Record

The fault record section describes how the flags, fault-data, and address-of-faulting-instruction fields of the fault record are used for the fault type and subtypes.

### Saved IP

The saved IP section describes what value is saved in the RIP register (r2) of the stack frame the processor was using when the fault occurred.

### Process State Changes

The process state changes section describes the effects that the fault subtypes have on the state of the process.



## Arithmetic Faults

**Fault Type:**

3<sub>16</sub>

**Fault Subtype:**

Number<sub>16</sub>

Name

0

Reserved

1

Integer Overflow

2

Arithmetic Zero-Divide

3-F

Reserved

**Function:**

Indicates that there is a problem with an operand or the result of an arithmetic instruction. This fault type applies only to ordinal and integer instruction, not floating-point instructions.

The integer-overflow fault occurs when the result of an integer instruction overflows the destination and the integer-overflow mask in the arithmetic-controls register is cleared. Here, the  $n$  least significant bits of the result are stored in the destination, where  $n$  is the destination size.

The arithmetic zero-divide fault occurs when the divisor operand of an ordinal or integer divide instruction is zero.

**Fault Record:**

**Flags:**

Not used.

**Fault Data:**

Not used.

**Addr. Fault. Inst.:**

IP for the instruction on which the processor faulted.

**Saved IP:**

IP for the instruction that would have been executed next, if the fault had not occurred.

**Proc. State Changes:**

A process-state change accompanies the integer-overflow fault, because the result is stored in the destination before the fault is generated. The faulting instruction can thus not be reexecuted.

A process-state change does not accompany the arithmetic zero-divide fault, because the fault occurs before the execution of the faulting instruction.



## Constraint Faults

<b>Fault Type:</b>	5 <sub>16</sub>		
<b>Fault Subtype:</b>	<b>Number</b> <sub>16</sub>	<b>Name</b>	<b>Number</b> <sub>16</sub>
	0	Reserved	0
	1	Constraint Range	1
	2	Invalid SS	2
	3-F	Reserved	3-F

**Function:** Indicates that the processor is either in or not in the required state for the instruction to be executed.

The constraint-range fault occurs when a fault-if instruction is executed and the condition code in the arithmetic controls matches the condition required by the instruction.

The invalid-SS fault occurs when an instruction attempts to reference a segment by means of an SS, when the processor is not in the supervisor mode.

**Fault Record:**

<b>Flags:</b>	Not used.
<b>Fault Data:</b>	Not used.
<b>Addr. Fault. Inst.:</b>	IP for the instruction on which the processor faulted

**Saved IP:** Not used.

**Proc. State Changes:** No process-state changes accompany either of these faults. For the constraint-range fault, the fault occurs after the fault-if instruction has been executed, but the instruction has no effect on the process state.

The invalid-SS fault occurs before the faulting instruction.

## Descriptor Faults

**Fault Type:**

D<sub>16</sub>

**Fault Subtype:**

Number<sub>16</sub>

Name

0

Reserved

Reserved

1

Invalid Descriptor

Invalid Descriptor

2-F

Reserved

Reserved

**Function:**

Indicates that an address or SS cannot be translated into a physical address because of an invalid segment-table entry.

The descriptor-invalid fault is the only one of this fault type. This fault occurs in either of two situations: (1) when an SS points to a segment descriptor that has an invalid type or (2) when an SS points to a segment descriptor that is an embedded type, but the descriptor is not being used in a semaphore operation.

**Fault Record:**

**Flags:**

Not used.

**Fault Data:**

The segment index for the invalid descriptor is stored in bits 5 through 31 of the second word of the fault-data field.

**Addr. Fault. Inst.:**

IP for the instruction on which the processor faulted.

**Saved IP:**

Same as the address-of-faulting-instruction field.

**Proc. State Changes:**

A process-state change does not accompany the invalid-descriptor fault, because the fault occurs before the execution of the faulting instruction.

## Event Faults

<b>Fault Type:</b>	E <sub>16</sub>		
<b>Fault Subtype:</b>	<b>Number</b> <sub>16</sub>	<b>Name</b>	<b>Number</b> <sub>16</sub>
	0	Reserved	0
	1	Event Notice	1
	2-F	Reserved	2-F

**Function:** Indicates that software has generated a fault event.

The event-notice fault is the only one of this fault type. This fault occurs in either of the following situations: (1) when a process is dispatched and the event-fault-request flags in the process's PCB are set, or (2) when a processor receives a process notice IAC and the event-fault-request flags are set for the process currently running on that processor.

**Fault Record:** **Flags:** Not used.

**Fault Data:** Not used.

**Addr. Fault. Inst.:** IP for the instruction on which the processor faulted.

**Saved IP:** IP for the instruction that would have been executed next, if the fault had not occurred.

**Proc. State Changes:** If this fault occurs while a process is being dispatched, the fault is generated before work on the process begins. This allows the fault handler to either never begin work on the process or to return to the process and begin work on it.

If this fault occurs while an instruction is being executed, the processor does one of the following: (1) terminates the instruction as if it had not yet begun execution, (2) completes execution of the instruction, or (3) suspends the instruction, saving the intermediate state in the resumption record. The instruction being executed determines which action is taken.

The process state thus may change in conjunction with the occurrence of this fault. However, when the state does change, the processor saves sufficient state information to allow the state of the process to be saved when the process is suspended or to allow resumption of the instruction on a return from the fault handler.

## Floating-Point Faults

### Fault Type:

4<sub>16</sub>

### Fault Subtype:

#### Bit Number

#### Name

Bit 0	Floating Overflow
Bit 1	Floating Underflow
Bit 2	Floating Invalid-Operation
Bit 3	Floating Zero-Divide
Bit 4	Floating Inexact
Bit 5	Floating Reserved-Encoding
Bits 6 and 7	Reserved

### Function:

Indicates that there is a problem with an operand or the result of a floating-point instruction. Each floating-point fault is assigned a bit in the fault-subtype field. Multiple floating-point faults can only occur simultaneously, however, with the floating-overflow, floating-underflow, and floating-inexact faults.

The floating-point faults are described in detail in the section in Chapter 7 titled "Exceptions and Fault Handling." The following paragraphs give a brief description of each floating-point fault.

A floating-overflow fault occurs when (1) the floating-point overflow mask is clear and (2) the infinitely precise result of a floating-point instruction exceeds the largest allowable finite value for the specified destination format. This fault interacts with the floating-inexact fault (as described in Chapter 7).

A floating-underflow fault occurs when (1) the floating-point underflow mask is clear and (2) the infinitely precise result of a floating-point instruction is less than the smallest possible normalized, finite value for the specified destination format. This fault interacts with the floating-inexact fault (as described in Chapter 7).

The floating invalid-operation fault occurs when (1) the floating-point invalid-operation mask is clear and (2) one of the source operands for a floating-point instruction is inappropriate for the type of operation being performed.

The floating zero-divide fault occurs when (1) the floating-point zero-divide mask is clear and (2) the divisor operand of a floating-point divide instruction is zero.

The floating-inexact fault occurs when (1) the floating-point inexact mask is clear and (2) an infinitely precise result cannot be encoded in the format specified for the destination operand. This fault interacts with the floating-overflow and floating-underflow faults (as described in Chapter 7).

The floating reserved-encoding fault occurs when a denormalized value is used as an operand in a floating-point instruction and the normalizing-mode bit in the arithmetic controls is clear.

<b>Fault Record:</b>	<b>Flags:</b>	<p><b>F0</b> — Used if inexact fault occurs in conjunction with overflow or underflow fault. If set, F0 indicates that the adjusted result has been rounded toward <math>+\infty</math>; if clear, F0 indicates that the adjusted result has been rounded toward <math>-\infty</math>.</p> <p><b>F1</b> — Used with overflow and underflow faults only. If set, F1 indicates that the adjusted result has been bias adjusted, because its exponent was outside the range of the extended-real format.</p>
	<b>Fault Data:</b>	Used only with overflow and underflow faults. Adjusted result is stored in this field in extended-real format (as shown in Figure 7-5).
	<b>Addr. Fault. Inst.:</b>	IP for the instruction on which the processor faulted
<b>Saved IP:</b>	IP for the instruction that would have been executed next, if the fault had not occurred.	
<b>Proc. State Changes:</b>	<p>Process-state changes accompany the floating-overflow, floating-underflow, and floating-inexact faults, because a result is stored in the destination before the fault is generated. The faulting instruction can thus not be reexecuted.</p> <p>Process-state changes do not accompany the floating invalid-operation, floating zero-divide, and floating reserved-encoding faults, because the faults occur before the execution of the faulting instruction.</p>	

## Machine Faults

<b>Fault Type:</b>	8 <sub>16</sub>		
<b>Fault Subtype:</b>	Number <sub>16</sub>	Name	Number <sub>16</sub>
	0	Reserved	0
	1	Bad Access	1
	2-F	Reserved	2
<b>Function:</b>	Indicates that the processor has detected a hardware or memory-system error.		
	The bad-access fault is the only one of this fault type. This fault occurs whenever an unrecoverable memory error occurs on a physical memory operation.		
<b>Fault Record:</b>	<b>Flags:</b>	Not used.	
	<b>Fault Data:</b>	Not used.	
	<b>Addr. Fault. Inst.:</b>	Not used.	
<b>Saved IP:</b>	Not used.		
<b>Proc. State Changes:</b>	This fault may occur at any time. When it does occur, the accompanying state of the process is undefined. As a result, the processor is not able to return predictably from the fault handler to the point in the process where the fault occurred.		
	If this fault occurs during an atomic operation, there is no guarantee that the locking mechanism the memory subsystem uses for synchronization is unlocked. This is a fatal condition.		



## Operation Faults

<b>Fault Type:</b>	$2_{16}$				$8_{16}$
<b>Fault Subtype:</b>	<b>Number</b> <sub>16</sub>	<b>Name</b>	<b>Name</b>	<b>Number</b> <sub>16</sub>	
	0	Reserved	Reserved	0	
	1	Bad Access	Invalid Opcode	1	
	2	Reserved	Reserved	2-F	
	3		Reserved		
	4		Invalid Operand		
	5 - F		Reserved		

**Function:** Indicates that the processor cannot execute the current instruction because of invalid instruction syntax or operand semantics.

The invalid-opcode fault occurs when the processor attempts to execute an instruction that contains an undefined opcode or addressing mode.

The invalid-operand fault occurs when the processor attempts to execute an instruction for which one or more of the operands have special requirements and one or more of the operands do not meet these requirements. This fault subtype is not generated on floating-point instructions.

**Fault Record:** **Flags:** Not used.

**Fault Data:** Not used.

**Addr. Fault. Inst.:** IP for the instruction on which the processor faulted.

**Saved IP:** Not used.

**Proc. State Changes:** A process-state change does not accompany the operation faults, because the faults occur before the execution of the faulting instruction.

## Process Faults

<b>Fault Type:</b>	C <sub>16</sub>				
<b>Fault Subtype:</b>	Number <sub>16</sub>	Name	Name	Bit Number	Bit
	0	Reserved	Reserved	Bit 0	
	1	Segment Length	Time Slice	Bit 1	
	2-F	Page Rights	Reserved	Bit 2	

**Function:** Indicates that the current state of a process prohibits the processor from continuing to work on it.

There is only one process fault subtype, the time-slice fault. This fault occurs when an end-of-time-slice event occurs and the time-slice-reschedule flag in the process-controls word is clear.

The intended action following this fault is for the fault handler to collect information on the current state of the process. The fault handler can then store this information in the PCB for the process and suspend the process. Or, as an alternative, the fault handler can return to the process and use the saved process-state and instruction-resumption information to continue executing the process.

**Fault Record:** **Flags:** Not used.  
**Fault Data:** Not used.

**Addr. Fault. Inst.:** IP for the instruction on which the processor faulted.

**Saved IP:** IP for the instruction that would have been executed next, if the fault had not occurred.

**Proc. State Changes:** Since this fault occurs while an instruction is being executed, it is often accompanied by a process-state change. However, when the state does change, the processor saves sufficient state information to allow the processor to resume work on the instruction on a return from the fault handler or to allow the state of the process to be saved when the process is suspended.

When the fault occurs, the processor does one of the following: (1) terminates the instruction as if it had not yet begun execution, (2) completes execution of the instruction, or (3) suspends the instruction, saving the intermediate state in the resumption record. The instruction being executed determines which action is taken.

## Protection Faults

**Fault Type:**

7<sub>16</sub>

**Fault Subtype:**

**Bit Number**

**Name**

Bit 0

Reserved

Reserved

Bit 1

Time Slice

Segment Length

Bit 2

Reserved

Page Rights

Bit 3-7

Reserved

Reserved

### Function:

Indicates that an instruction has attempted to violate the addressing-protection rules of the processor architecture. Each protection fault is assigned a bit in the fault-subtype field. Both protection faults can occur simultaneously, in which case, the bits for both faults are set.

The segment-length fault can occur in either of the two following situations: (1) when an address operand in an instruction falls beyond the defined boundaries of a region, or (2) when the segment index within an SS is greater than the last entry in the segment table.

The page-rights fault occurs when the following two situations both occur: (1) an address operand references a page in a paged or bipaged region and (2) the page-table-directory entry or page-table entry associated with the reference page does not have the necessary page rights for the current execution mode of the processor.

The action that the processor takes when these faults occur allows the fault handler to modify the segment table, page-table-directory, or page-table when appropriate to correct the fault condition, then resume work on the process from the point where the fault occurred.

### Fault Record:

#### Flags:

**F0** — Used with page-rights fault only. If set, F0 indicates that an attempted write operation caused the fault; if clear, F0 indicates that an attempted read operation caused the fault.

**F1** — Not used.

#### Fault Data:

For a page-rights fault, the first two words of the fault-data field specify the page that was being accessed when the fault occurred. The 20 most-significant bits of the address that caused the fault are stored in bits 12 through 31 of the first word. (The two most-significant bits of the address are set to 0, which means that the processor interprets the value as an offset into a region). The segment index associated with the region that contains the address is stored in bits 5 through 31 of the second word.

For a segment-length fault where an address in the faulting instruction is beyond the specified size of the segment or region, the page that the address is trying to reference is specified in the first two words of the fault-data field as described for the page-rights fault.

For a segment-length fault where a segment index given in the faulting instruction is greater than the last index in the segment table, the segment index is given in the second word of the fault-data field as described for the page-rights fault.

**Addr. Fault. Inst.:** IP for the instruction on which the processor faulted.

**Saved IP:** Same as the address-of-faulting-instruction field.

**Proc. State Changes:** A process-state change accompanies each of the protection faults; however, sufficient state information is saved to permit either reexecution or completion of the faulting instruction on a return from the fault handler.

These faults occur while the faulting instruction is being executed. When the fault occurs, the processor will either (1) terminate the instruction as if it had not yet begun execution or (2) suspend the instruction, saving the intermediate state in the resumption record. The instruction being executed determines which action is taken.

Fault Record:	Flags:	Not used.
Saved IP:	Fault Data:	Not used.
Proc. State Changes:	Addr. Fault. Inst.:	IP for the instruction on which the processor faulted
Not used.		

**Structural Faults****Fault Type:**9<sub>16</sub>**Fault Subtype:**Number<sub>16</sub>**Name**

0	Reserved
1	Control
2	Dispatch
3	IAC
4-F	Reserved

**Function:**

Indicates that the state of one of the architecture-defined data structures is preventing the processor from performing a system operation. Examples of things that can cause a structural fault include a pointer in one data structure to a non-existent data structure or invalid state information in a data-structure field. These faults often occur while the processor is performing an internal (implicit) operation and may not be related to a particular instruction.

The control fault occurs either when (1) the invalid contents of a data structure are preventing a fault or interrupt from being handled or when (2) a fault occurs during the process of invoking an interrupt handler.

The dispatch fault occurs when the invalid contents of a data structure are preventing a process-dispatching action from being performed.

The IAC fault occurs when the invalid contents of a data structure are preventing an IAC from being executed.

**Fault Record:**

**Flags:** Not used.

**Fault Data:** Not used.

**Addr. Fault. Inst.:** IP for the instruction on which the processor faulted

**Saved IP:**

Not used.

**Proc. State Changes:**

When a structural fault occurs, the accompanying state of the process is undefined. The processor is thus not able to return predictably from the fault handler to the point in the process where the fault occurred. This condition is usually fatal.

**Trace Faults****Fault Type:** $1_{16}$ **Fault Subtype:****Bit Number****Name**

Bit 0

Reserved

Bit 1

Instruction Trace

Bit 2

Branch Trace

Bit 3

Call Trace

Bit 4

Return Trace

Bit 5

Prereturn Trace

Bit 6

Supervisor Trace

Bit 7

Breakpoint Trace

**Function:**

Indicates that the processor has detected one or more trace events. The processor's event tracing mechanism is described in detail in Chapter 16.

A trace event is the occurrence of a particular instruction or type of instruction in the instruction stream. The processor recognizes seven different trace events (instruction, branch, call, return, prereturn, supervisor, and breakpoint). It detects these events, however, only if a mode bit is set for the event in the process trace-controls word, which is cached in the processor chip. If, in addition, the trace-enable flag in the process controls is set, the processor generates a fault when a trace event is detected.

The fault is generated following the instruction that causes a trace event (or prior to the instruction for the prereturn trace event).

The following trace modes are available:

- **Instruction** — Generate trace event following any instruction.
- **Branch** — Generate trace event following any branch instruction when branch is taken. (Does not occur on branch and link and call instructions.)
- **Call** — Generate trace event following any call or branch-and-link instruction, or implicit procedure call (i.e., call to fault or interrupt handler).
- **Return** — Generate trace event following any return instruction.
- **Prereturn** — Generate trace event prior to any return instruction, providing the prereturn-trace flag in r0 is set. (The processor sets this flag automatically when prereturn tracing is enabled.)
- **Supervisor** — Generate trace event following any call-system instruction that references a supervisor procedure entry in a procedure table.



- **Breakpoint** — Generate trace event following any processor action that causes a breakpoint condition (such as a **mark** or **fmark** instruction).

There is a trace fault subtype and a bit in the fault-subtype field associated with each of these modes. Multiple fault subtypes can occur simultaneously, with the fault-subtype bit set for each subtype that occurs.

When a fault type other than a trace fault occurs during the execution of an instruction that causes a trace event, the non-trace-fault is handled before the trace fault. An exception to this rule is the prereturn trace fault. The prereturn trace fault will occur before the processor has a chance to detect a non-trace-fault, so it is handled first.

Likewise, if an interrupt occurs during an instruction that causes a trace event, the interrupt is serviced before the trace fault is handled. Again, the prereturn trace fault is an exception. Since it occurs before the instruction, it will be handled before any interrupt that might occur during the execution of the instruction.

#### Fault Record:

**Flags:** Not used.

**Fault Data:** Not used.

**Addr. Fault. Inst.:** IP for the instruction that caused the trace event, except for the prereturn trace fault. For the prereturn trace fault, this field has no defined value.

#### Saved IP:

IP for the instruction that would have been executed next, if the fault had not occurred.

#### Proc. State Changes:

A process state change accompanies all the trace faults (except the prereturn trace fault), because the events that can cause a trace fault occur after the faulting instruction is completed. As a result, the faulting instruction cannot be reexecuted upon returning from the fault handler.

Since the prereturn trace fault occurs before the **ret** instruction is executed, a process state change does not accompany this fault and the faulting instruction can be executed upon returning from the fault handler.

Type Faults

Fault Type:	A <sub>16</sub>		
Fault Subtype:	Number <sub>16</sub>	Name	Number <sub>16</sub>

0	Reserved	0	Reserved
1	Type Mismatch	1	Type Mismatch
2	Contents	2	Contents
3	Reserved	3	Reserved

**Function:** Indicates that the contents of an architecture-defined data structure or its descriptor are inconsistent with the operation that the processor is trying to perform.

The type-mismatch fault occurs when the type information in a segment descriptor does not match the operation the processor is being asked to perform. For example, a type-mismatch fault occurs when the SS given in a resume-process instruction (**resumprcs**) does not point to a PCB segment.

The contents fault occurs when the information in a segment is not defined or is inconsistent.

**Fault Record:** **Flags:** Not used.

**Fault Data:** Not used.

**Addr. Fault. Inst.:** IP for the instruction on which the processor faulted

**Saved IP:** Not used.

**Proc. State Changes:** When a type fault occurs, the accompanying state of the process is undefined. The processor is thus not able to return predictably from the fault handler to the point in the process where the fault occurred.

## Virtual-Memory Faults

<b>Fault Type:</b>	6 <sub>16</sub>		
<b>Fault Subtype:</b>	<b>Number</b> <sub>16</sub>	<b>Name</b>	<b>Number</b> <sub>16</sub>

0	Reserved	Reserved	0
1	Type Mismatch	Invalid Segment-Table-Entry	
2	Contents	Invalid Page-Table-Directory-Entry (PTDE)	
3	Reserved	Invalid Page-Table-Entry (PTE)	
4-F		Reserved	

**Function:** Indicates that an address or an SS in an instruction cannot be translated into a physical address, because the segment or page being referenced is not in physical memory.

The invalid-segment-table-entry fault occurs when the valid flag in a segment descriptor is 0, which can mean that the segment, the page-table directory, or the page table that the segment descriptor points to is not in physical memory.

The invalid-PTDE fault occurs when the valid flag in a page-table-directory entry is 0, which means that the page table that the entry points to is not in physical memory.

The invalid-PTE fault occurs when the valid flag in a page-table entry is 0, which means that the page that the entry points to is not in physical memory.

The action that the processor takes when these faults occur allows the fault handler to copy the missing segment or page from the disk into physical memory, then resume work on the process from the point where the fault occurred.

**Fault Record:** **Flags:** Not used.

**Fault Data:** For an invalid-PTE or invalid-PTDE fault, the 20 most-significant bits of the address that the instruction faulted on is stored in bits 12 through 31 of the first word of the fault data field.

For an invalid-segment-table-entry fault or for an invalid-PTE fault for a large-segment table, the segment index is stored in bits 5 through 31 of the second word of the fault-data field.

**Addr. Fault. Inst.:** IP for the instruction on which the processor faulted

**Saved IP:** Same as the address-of-faulting-instruction field.

**Proc. State Changes:** A process-state change accompanies each of the virtual-memory faults, however, sufficient state information is saved to permit either

reexecution or completion of the faulting instruction on a return from the fault handler.

These faults occur while the faulting instruction is being executed. When the fault occurs, the processor will either (1) terminate the instruction as if it had not yet begun execution or (2) suspend the instruction, saving the intermediate state in the resumption record. The instruction being executed determines which action is taken.









## CHAPTER 13 PROCESS MANAGEMENT

This chapter introduces the 80960MC processor's process-management facilities. Included is a discussion of process-management concepts, the process-control block (PCB), and the requirements for managing a single process. Chapter 14 describes the management of multiple processes.

### PROCESS-MANAGEMENT OVERVIEW

The processor provides a set of low-level and high-level process-management facilities. With these tools, the kernel or system-executive is able to efficiently allocate processor resources to one or more processes, using any of a wide variety of process-management techniques.

The following section provides an overview of these process-management facilities.

#### Process Structure

A process is a unit of work that the processor can schedule, dispatch, and execute. It can be used to execute an application task, a kernel utility, or a monitor command shell.

A process is made up of two parts: an address space and a PCB. The address space contains the code, stacks, static data, and heap data for the process. When the processor's virtual-addressing mode is being used, the address space for a process consists of three process-specific regions (0, 1, and 2). When the process is bound to the processor for execution, these regions are joined with region 3, which is shared by all processes, to form the process-execution address space (or process address space) for the process. When the physical-addressing mode is being used, the address space for a process consists of all of physical memory.

The PCB defines the address space for the process and provides a repository of state information for the process. In a multitasking system, the PCB also provides a device for scheduling and dispatching multiple processes.

#### Process State

The following items define the state of a process at any given time:

- The address-space image
- The state of the global registers
- The state of the stack, including the local registers
- The state of the arithmetic controls
- The state of the process controls

- The state of the trace controls

When a process is bound to the processor, the state of the process is contained within the processor, the address space, and the PCB. When a process is suspended, the state of the process is contained in the address space and the PCB. The PCB and region mechanism allows a processor to work on several processes concurrently, merely by switching among PCBs and address spaces.

## Using Processes

The process-management facilities of the processor support single-process systems, multitasking systems, and multiprocessor systems. In a single-process system, a process is bound to the processor at initialization time. The processor then executes this process alone. This single process can be used to support a dedicated or embedded activity or to run a user-defined, process-management mechanism. In the latter case, the user-defined processes are transparent to the processor.

The processor provides several mechanisms for managing a multitasking system. These mechanisms can be roughly divided into two categories: explicit process-dispatching and self-dispatching. When using explicit dispatching, the kernel binds a process to the processor or suspends a process by means of explicit instructions to the processor.

The processor also provides a set of high-level process-management facilities that allow processes to be dispatched automatically, independently from the activity of the kernel. This self-dispatching mechanism makes use of a system-defined dispatch port, which the processor uses to schedule and dispatch processes.

In a multiprocessor system, these high-level process-management facilities greatly simplify the allocation of processor resources to ready and executing processes.

## PROCESS-CONTROL BLOCK

The PCB defines a process for the processor. It specifies the execution environment for the process, provides a place to record the execution status of the process, and maintains information about the system resources that have been allocated to the process.

Figure 13-1 shows the structure of the PCB and Figure 13-2 shows the structure of the process-controls word, which is one of the fields of the PCB. The following paragraphs describe the function of each field of the PCB.

- The state of the process controls
- The state of the arithmetic controls
- The state of the stack, including the local registers
- The state of the global registers
- The address-space image

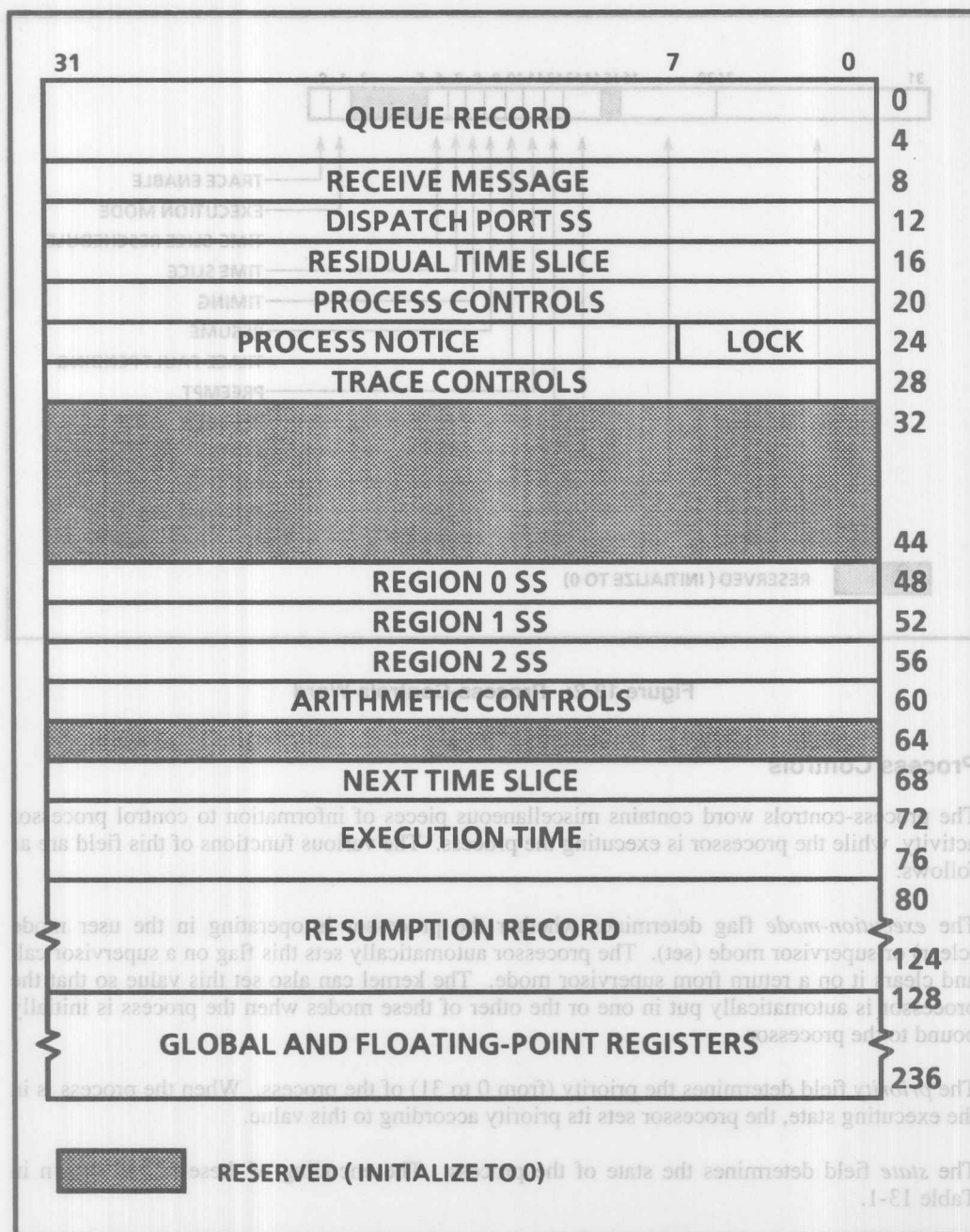


Figure 13-1: Process-Control Block (PCB)

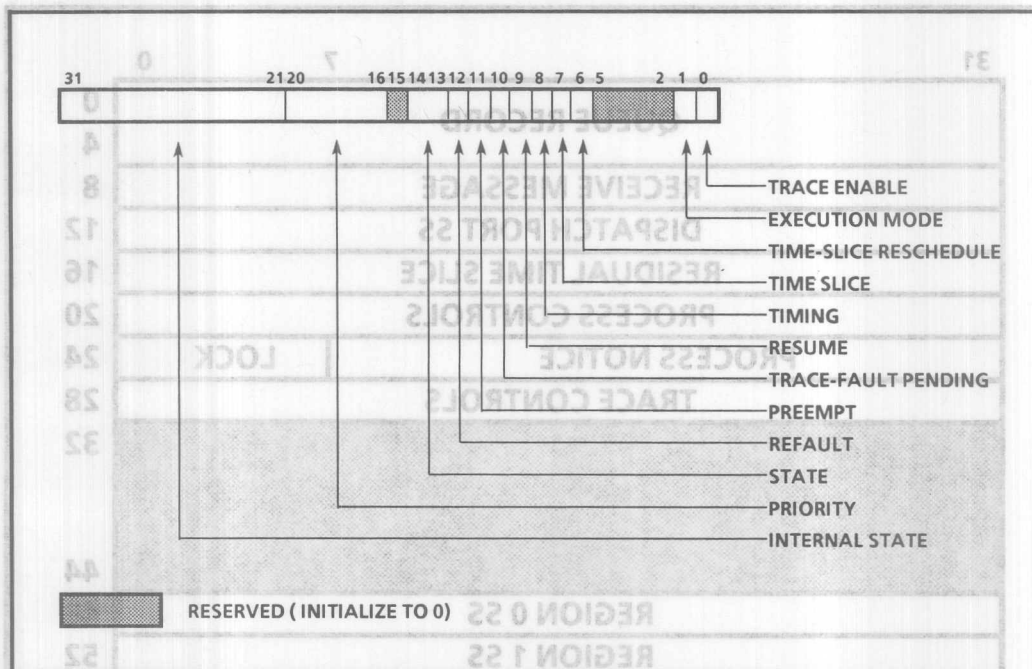


Figure 13-2: Process-Controls Word

### Process Controls

The process-controls word contains miscellaneous pieces of information to control processor activity, while the processor is executing the process. The various functions of this field are as follows:

The *execution-mode* flag determines whether the processor is operating in the user mode (clear) or supervisor mode (set). The processor automatically sets this flag on a supervisor call and clears it on a return from supervisor mode. The kernel can also set this value so that the processor is automatically put in one or the other of these modes when the process is initially bound to the processor.

The *priority* field determines the priority (from 0 to 31) of the process. When the process is in the executing state, the processor sets its priority according to this value.

The *state* field determines the state of the process. The encoding of these bits is shown in Table 13-1.

These bits tell the processor or software whether the process either

- has been interrupted so the processor can service an interrupt ( $01_2$ ), or
- is currently being executed or waiting to be executed ( $00_2$ ).

Table 13-1: Encoding of the Process-State Field

State Field	Process State
00	Executing, ready, or blocked
01	Interrupted
10	Reserved
11	Reserved

The *timing*, *time-slice*, and *time-slice-reschedule* flags control the timing and time-slice scheduling of processes. This subject is discussed in Chapter 14 in the section titled "Time-Slice Scheduling."

The *preempt* flag determines whether or not a process is eligible to preempt another processes. Process preemption is described in Chapter 14 in the section titled "Process Preemption."

The *resume* flag signals the processor that an instruction has been suspended. The processor sets this flag whenever it suspends an instruction to handle an interrupt or fault. On a return from the interrupt or fault handler, the processor checks this flag and performs an instruction resumption action if the flag is set.

The *refault* flag is used in conjunction with the handling of override faults. When an override fault is detected, the processor sets this flag. On a return from an override-fault handler, the processor checks this flag and refaults on the original fault (the one that occurred before the override fault). Further discussion of this flag is provided in Chapter 12 in the sections titled "Override Call/Return Action" and "Refault Operation."

The *trace-enable* and *trace-fault-pending* flags control tracing. The *trace-enable* flag determines whether trace faults are to be generated (set) or not-generated (clear). The processor uses the *trace-fault-pending* flag to determine if a trace event has been detected (set) or not (clear). The use of these flags are discussed in detail in Chapter 16.

Bits 2 through 5, 15, and 21 through 31 are reserved. These bits should be set to 0 when the PCB is created and not altered after that.

The kernel can alter the settings of the process-controls bits in several ways, as described later in this chapter in the section titled "Changing the Process-Controls."

## Process-State Fields

Several fields are provided in the PCB for storing the state of the process. These fields define the state of the process when the process is bound to the process and provide a place to store state information when the process is suspended.

The *arithmetic-controls* field contains the state of the arithmetic controls.



The *trace-controls* field contains the state of the trace controls.

The *region 0 SS*, *region 1 SS*, and *region 2 SS* fields contain SS's for the three process-specific regions of the process's address space.

The *global and floating-point registers* field provides a place to store the state of the global and floating-point registers when a process is suspended. The kernel should not normally access these fields except to clear them when the PCB is created. Also, on creation of the PCB, a pointer to the base of the local stack must be placed in bytes 236 through 239 (global register g15).

The *resumption-record* field provides storage space for instruction resumption information. If an instruction is suspended to handle an interrupt or a fault, the resumption record for the instruction is copied into this field when the processor returns from the interrupt or fault handler. Refer to the section in Chapter 10 titled "Servicing an Interrupt" and the section in Chapter 12 titled "Fault-Handling Action" for further discussion of the use of the resumption-record field.

### Process Scheduling and Communication Fields

The following fields are provided to support the processor's high-level process management facilities. These fields are not used when the processor is operated in a single-process application.

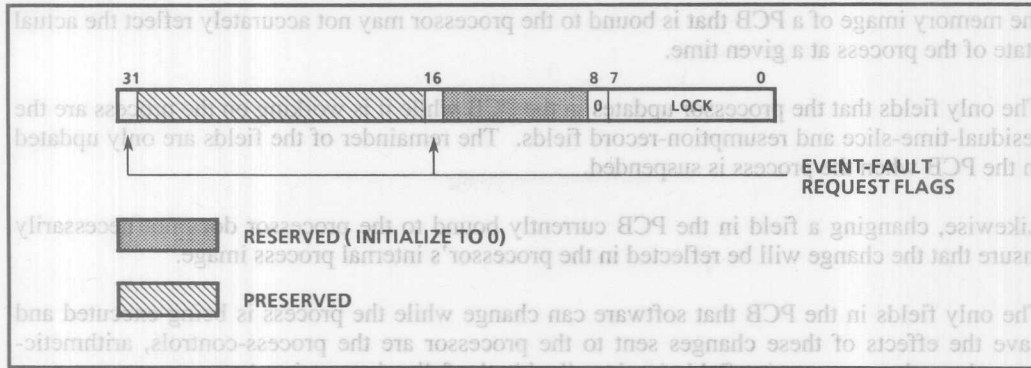
The *dispatch-port-SS* field provides an SS pointer to the dispatch port that the process is to be queued to when the process is suspended.

The *queue-record* field allows several PCBs to be linked together to form a queue. Processes are typically queued to ports or semaphores. The structure of the queue-record field is given in Chapter 14 in the section titled "Queue Record."

The processor provides a means of passing 1-word messages between processes. The *receive-message* field provides a temporary storage location for such messages. The message passing mechanism is described in Chapter 14 in the section titled "Interprocess Communication."

The *lock* field allows the processor to lock the PCB for the process it is working on, by setting bit 0 of the lock to 1. This lock supports multiprocessor systems. It provides a means for one processor to determine if another processor is currently working on a process, by reading the lock in the process's PCB. This lock does not prevent a processor from reading or altering a PCB. It merely acts as a flag to show whether or not a process is currently bound to a processor.

The *process-notice* field consists of two flags at bits 16 and 31 (as shown in Figure 13-3). (Bits 17 through 30 are available to software.) If these event-fault-request flags are set, the processor signals an event-notice fault either (1) when a processor attempts to dispatch the process, or (2) when the process is already bound to a processor and the processor receives a check-process-notice IAC from another processor. This field is cleared when an event-notice fault is generated.



**Figure 13-3: Process Notice Field and Event-Fault Flags**

The process-notice field also supports multiprocessor systems. It offers a means for one processor to preempt a process running on another processor or for a processor to dequeue a process from a dispatching port. (Note that this is only one of the methods that the processor provides for preempting a process.) Further discussion of the process-notice field is provided in Chapter 12 in the section titled "Event-Notice Fault."

### Process-Timing Fields

The processor provides facilities for counting the amount of time that a processor spends working on a process. It also provides facilities for scheduling multiple processes on the basis of time slices. The following fields support these facilities. The use of these fields are discussed in detail in Chapter 14 in the section titled "Time-Slice Scheduling."

The processor uses the *execution-time* field to keep a running count of the amount of time the process has spent in the execution state. The field contains a long-ordinal value (64 bits). The count saved in this field is in units of ticks. The processor updates this field in the PCB at the end of each time slice.

The processor uses the *next-time-slice* and *residual-time-slice* fields for time-slice scheduling. The next-time-slice field contains an ordinal value (32 bits) that gives the software preset time (in ticks) that the processor is allowed to work on the process before a time-slice event is generated. The processor keeps a count of the time remaining in the current time-slice in the residual-time-slice field (which also contains an ordinal value).

Refer to the section in Chapter 14 titled "Process Timing" for a detailed discussion of how the processor uses the execution-time, next-time-slice, and residual-time-slice fields for process timing.

### Storing of PCB Fields in the Processor

When a process is bound to the processor, certain fields from the PCB are copied into the processor and altered as the state of the process changes. When the processor alters an internally held field of the PCB, it does not generally update the field in memory. As a result,

the memory image of a PCB that is bound to the processor may not accurately reflect the actual state of the process at a given time.

The only fields that the processor updates in the PCB while it is working on the process are the residual-time-slice and resumption-record fields. The remainder of the fields are only updated in the PCB when the process is suspended.

Likewise, changing a field in the PCB currently bound to the processor does not necessarily insure that the change will be reflected in the processor's internal process image.

The only fields in the PCB that software can change while the process is being executed and have the effects of these changes sent to the processor are the process-controls, arithmetic-controls, and process-notice fields (as described in the following sections).

#### NOTE

At initialization, all the fields of the processor's internal process-controls image are set to 0 except execution mode, which is set to 1 (supervisor mode).

### Changing the Process Controls

The kernel can change the process controls for the current process using any of the following three methods:

- Modify-process-controls instruction (**modpc**)
- Alter the saved process controls prior to a return from an interrupt handler
- Alter the saved process controls prior to a return from a fault handler

The **modpc** instruction reads and modifies the process controls cached in the processor. (It does not change the process controls word in the PCB for the current process.)

In the latter two methods, the kernel changes the process controls in the interrupt or fault record that is saved on the stack. On the return from the interrupt or fault handler, the modified process controls are copied into the processor's internal process controls.

Two things should be noted with regard to modifying the saved process controls. First, this technique for changing the process controls can be used on a fault only if the fault handler was invoked by means of an implicit supervisor call. Second, the saved process controls are only copied into the process-controls image contained in the processor; the process controls in the PCB are not affected.

When the process controls are changed as described above, the processor acts on the changes as soon as it receives the new information, except for the following situations.

There is no guarantee that the processor will act on the change to the process-state field. The only case where such a change will have the desired result is if the process state in the saved-process controls is changed to "executing" prior to a return from an interrupt handler.

Changing the resume flag can cause the execution of the subsequent instruction to yield unpredictable results.

If the **modpc** instruction is used to change the trace-enable flag, the processor does not guarantee to act on the change until after up to four more instructions have been executed.

### Changing the Arithmetic Controls

The kernel or an applications program can change the arithmetic controls using the **modac** instruction. This instruction only affects the internally cached arithmetic controls. The arithmetic controls word in the PCB for the current process is not changed.

### Changing the Process-Notice Field

The process-notice field of the PCB is not cached in the processor. However, the check process-notice IAC can be used to cause the processor to check the process-notice field in the PCB for the currently running process. (Refer to the discussion of the process-notice field earlier in this chapter in the section titled "Process Scheduling and Communication Fields.")

## REQUIRED SOFTWARE SUPPORT FOR A SINGLE-PROCESS SYSTEM

Figure 9-1 shows the system-data structures required to support a single-process system. Note that the single process is defined by means of a PCB and the three process-specific regions of the address space.

Figure 13-4 shows the required fields of the PCB for a single-process system. The PCB in this application is used primarily to contain initialization information. Once the process is bound to the processor at initialization, the only field of the PCB that the processor will use is the resumption record field.

Also, in single-process applications, the timing flag in the process-controls word should be set to 0, to disable timing.

### PHYSICAL ADDRESSING VERSES VIRTUAL ADDRESSING

If the processor is going to execute the process using strictly the physical-addressing mode, the range pointers in the PCB are not required.

However, an SS and a process-segment descriptor for the PCB are required, with the base address in the segment descriptor aligned to a 64-byte boundary and pointing to the first byte in physical memory of the PCB.

Figure 13-4: Process-Control Block for Single-Process System

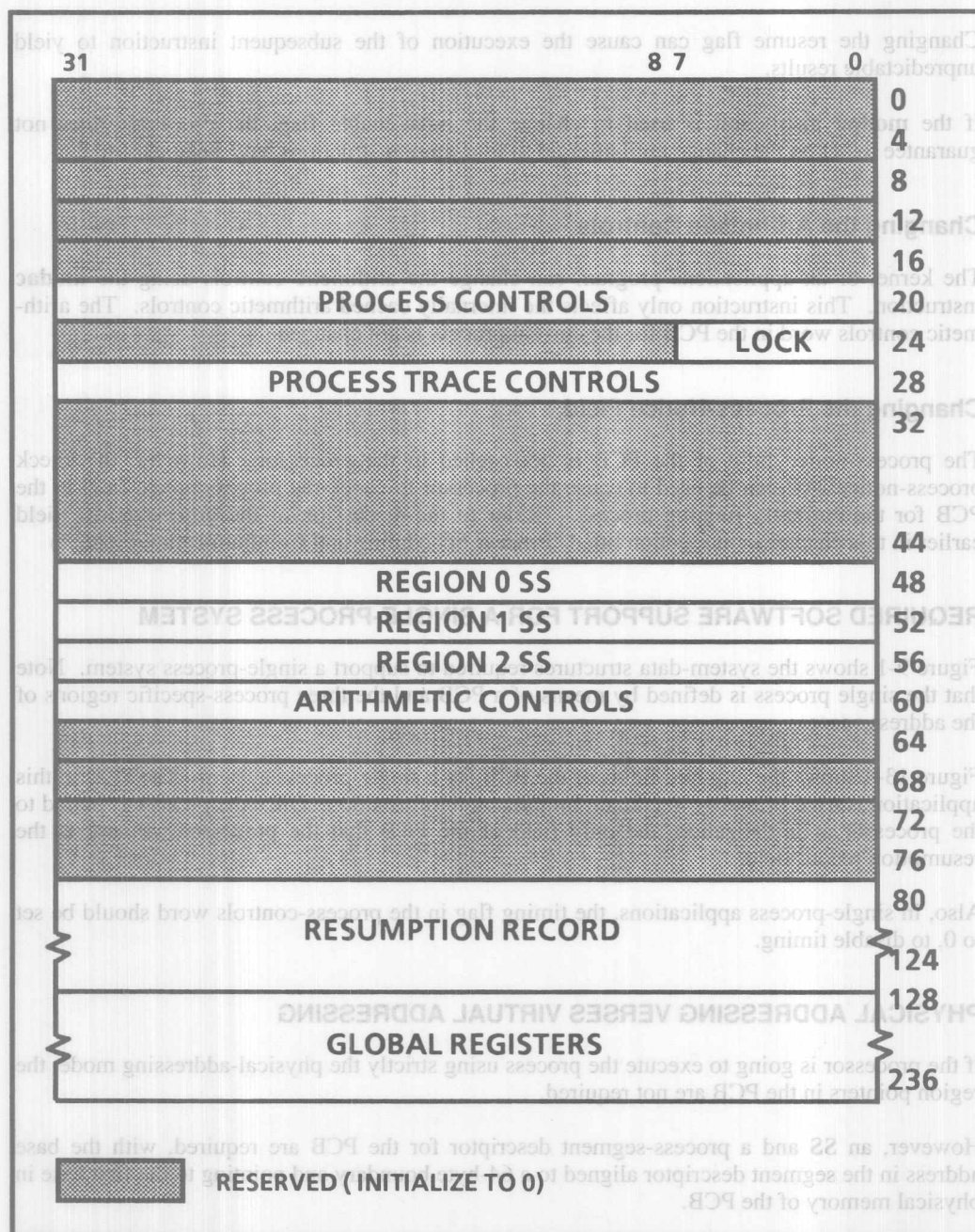


Figure 13-4: Process-Control Block for Single-Process System



## PROCESS HANDLING IN A SINGLE-PROCESS SYSTEM

In a single-process system, the initialization code must bind the process to the processor using a resume-process instruction (**resump<sub>prcs</sub>**). Once this binding is complete, the processor works on this process until the processor is either shut down or placed in the stopped state.

For diagnostic purposes, the PCB fields that are held internally in the processor can be written out to the PCB in memory using the save process instruction (**savep<sub>prcs</sub>**).



## PROCESS HANDLING IN A SINGLE-PROCESS SYSTEM

In a single-process system, the initialization code must bind the process to the processor using a resume-process instruction (resmpres). Once this binding is complete, the processor works on this process until the processor is either shut down or placed in the stopped state.

For diagnostic purposes, the PCB fields that are held internally in the processor can be written out to the PCB in memory using the save process instruction (savepres).

---

# *Multiple-Process Management*

---

**14**



## CHAPTER 14 MULTIPLE-PROCESS MANAGEMENT

This chapter discusses the facilities that the processor provides to manage multiple processes in multitasking systems. Included are descriptions of the process management tools provided for explicit process dispatching, self-dispatching, process synchronization, and interprocess communication.

### OVERVIEW OF MULTIPLE-PROCESS-MANAGEMENT FACILITIES

The process management facilities described in this chapter, and in Chapters 13 and 15, provide a general set of tools for designing a wide variety of process management mechanisms. In showing how these facilities can be used to support multitasking kernels, three general process-management scenarios are presented:

- A completely software-implemented system that runs within the context of a single 80960MC-defined process.
- A largely software-implemented system that uses several of the processor's low-level process management tools to explicitly schedule and dispatch multiple processes.
- A partly software-implemented system that uses the processor's high-level process management tools for automatic scheduling and dispatching of multiple processes, process synchronization, and interprocess communication.

The process management tools to support the first scenario are described in Chapter 13.

The tools to support the second and third techniques are described in this chapter. These tools are divided into two groups: low-level tools and high-level tools.

The low-level tools include the following:

- The PCB presented in Chapter 13, which allows a kernel to define a process and bind it to the processor for execution.
- Two process-handling instructions that permit the kernel to explicitly bind a process to the processor or suspend work on the process.
- A process timing mechanism that provides the kernel with a method of scheduling multiple processes on the basis of time slices.

The high-level tools include the following:

- A dispatch port data structure that supports automatic scheduling and dispatching of processes.
- Semaphore and communication port data structures that allow synchronization of interacting processes.
- Message-passing facilities that permit messages to be passed among processes.

These high-level tools are a unique feature of the 80960MC architecture. They provide silicon-based support for several advanced process-management mechanisms.

## PROCESS MANAGEMENT CONCEPTS

This section presents several process management concepts that will help you in understanding the functions of and the actions taken by the low-level and high-level process management tools.

### Scheduling and Dispatching

The concepts of *scheduling* and *dispatching* are central to the development of process management schemes. Dispatching is the activity of assigning a process to a processor. Scheduling is the activity of maintaining a list of processes that are waiting to be dispatched. In designing a process management system, the major goal of the dispatching mechanism is to deploy processor resources rapidly, whereas the major goal of the scheduling mechanism is to provide efficient allocation of the processor resources to the executable processes.

### Process States

Once the kernel has created a process to run on the 80960MC processor, it will always be in one of the following states:

- Executing
- Interrupted (but executing)
- Ready
- Blocked

Figure 14-1 shows the relationship of these states.

In the executing state, the process is bound to the processor and is being executed. Being bound to the processor means that the processor has read the contents of the process's PCB, and knows the location of the address-space regions for the processor. It has also copied process-state information, such as the process controls and arithmetic controls, from the PCB into internal registers or buffers.

Only one process can be bound to the processor at a time. In general, a processor should not be instructed to bind itself to another process until it has first suspended the current process. To suspend a process, the processor copies the parts of the process's PCB that it holds internally back into the PCB in memory, so that the PCB accurately defines the state of the suspended process. The save process (**saveprcs**) and resume process (**resumprrcs**) instructions cause the processor to explicitly save a process or bind itself to a process. When using self-dispatching, the processor performs these tasks automatically. (The **saveprcs** and **resumprrcs** instructions and self-dispatching of processes are described later in this chapter.)

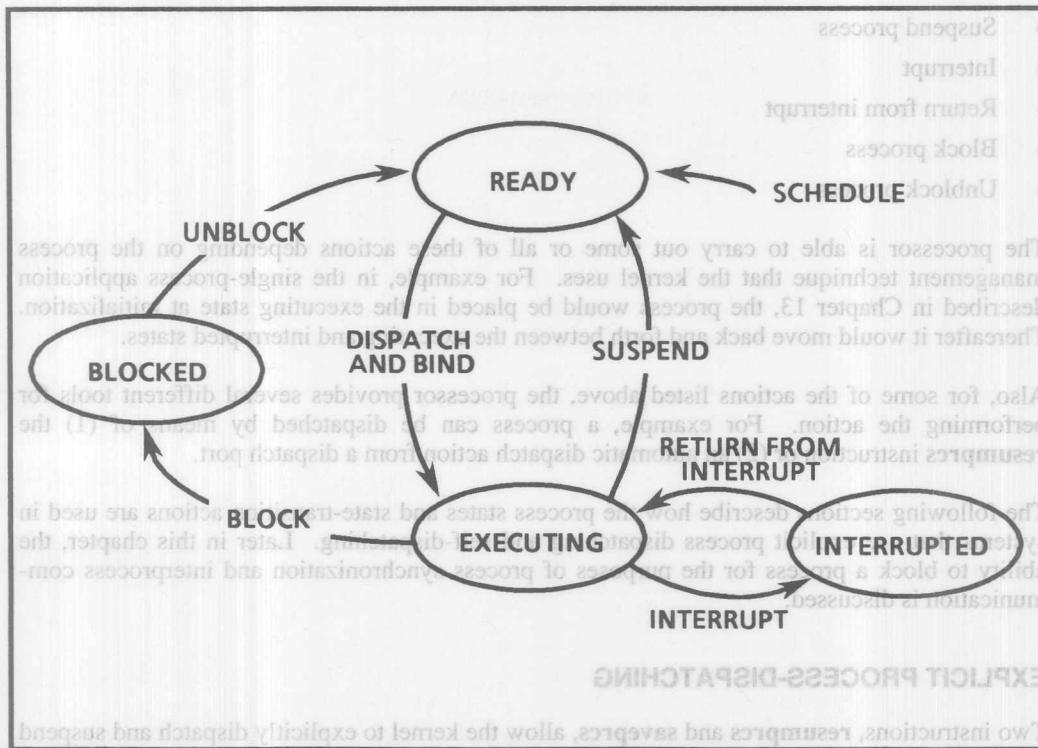


Figure 14-1: Process States

While in the executing state, a process can be interrupted. In the interrupted state, the process remains bound to the processor, but the processor is executing an interrupt-handler procedure.

A process is in the ready state when a PCB exists for the process and the PCB is enqueued on a dispatch port. A process is also said to be in ready if it is available to be bound to the processor using the **resumprcs** instruction. A process in the ready state is suspended.

The blocked state is used with the processor's process-synchronization and message-passing mechanisms. A process is blocked either when it is enqueued to a semaphore (waiting to receive a signal) or when it is enqueued to a communication port (waiting to receive a message). A process in the blocked state is suspended.

### State-Transition Actions

To aid in managing multiple processes, the processor supports a set of actions that moves processes among the four possible process states. These actions are listed below.

- Schedule process
- Dispatch and bind process



- Suspend process
- Interrupt
- Return from interrupt
- Block process
- Unblock process

The processor is able to carry out some or all of these actions depending on the process management technique that the kernel uses. For example, in the single-process application described in Chapter 13, the process would be placed in the executing state at initialization. Thereafter it would move back and forth between the executing and interrupted states.

Also, for some of the actions listed above, the processor provides several different tools for performing the action. For example, a process can be dispatched by means of (1) the **resumprrcs** instruction or (2) an automatic dispatch action from a dispatch port.

The following sections describe how the process states and state-transition actions are used in systems that use explicit process dispatching and self-dispatching. Later in this chapter, the ability to block a process for the purposes of process synchronization and interprocess communication is discussed.

## EXPLICIT PROCESS-DISPATCHING

Two instructions, **resumprrcs** and **saveprcs**, allow the kernel to explicitly dispatch and suspend processes, respectively. These instructions perform similar functions to the RESUME and SAVE functions provided in most UNIX<sup>TM</sup> kernels.

The **resumprrcs** instruction takes a process in the ready state and binds it to the processor, at which time the processor begins executing the process. Here, the process is considered in the ready state if a PCB has been created for the process and a segment descriptor for the PCB exists in the segment table.

The **saveprcs** instruction causes the processor to write any internally held parts of the PCB out to the PCB in memory. Following the execution of this instruction, the process is still bound to the processor, but the state of the PCB in memory is like it would be if the process had just been suspended. A **resumprrcs** instruction can then be safely executed to bind a new process to the processor.

It should be noted that **resumprrcs** and **saveprcs** instructions are tools to assist the kernel in dispatching and suspending processes, but they do not do the whole job. These instructions will most often be used in a fault- or interrupt-handler procedure, in which case the kernel will need to modify the PCB for the suspended process in between the **saveprcs** instruction to suspend the current process and the **resumprrcs** instruction to dispatch the next process. This work involves changing the PCB to reflect the state of the process prior to invoking the fault or interrupt handler. This can often be done merely by changing the frame pointer in the saved global registers and the process-state bits in the process controls.

## PROCESS TIMING

The processor provides an on-chip counter that can be used for both process and idle timing. When used for process timing, the counter's primary function is to support time-slice scheduling. It can be used, however, strictly to count execution time, as described at the end of this section.

The counter counts ticks. The time interval of a tick is described in Chapter 9 in the section titled "Processor Timing."

### Time-Slice Scheduling

With time-slice scheduling, the processor works on each process for a set duration, called a time-slice. When the processor begins work on a newly bound process, it begins counting. At the end of the time-slice, it generates a time-slice event, which either causes a time-slice fault to be signaled or causes the current process to be suspended and another process dispatched.

Six fields in the PCB support time-slice scheduling: the residual-time-slice, next-time-slice, and execution-time fields; and the timing, time-slice, and time-slice-reschedule flags in the process controls. These fields are used as follows.

The timing flag (if set) enables the timing function. If this flag is clear, the processor does not perform process timing. The **modpc** instruction can be used to toggle the timing flag, turning timing on and off. Also, the processor automatically clears this flag when it invokes an interrupt handler and restores the flag on the return from the handler. This action causes process timing to be turned off while the processor is servicing an interrupt.

The next-time-slice field determines the duration of a time-slice for the process. Each process can have a different time-slice value, ranging from  $2^{32} - 1$  ticks.

The residual-time-slice field is used to count the remaining time for the current time-slice. When the process is initially bound to the processor, the next-time-slice and residual-time-slice fields are the same. As the processor counts (while working on the process), it decrements the residual-time-slice field. When this field reaches 0, the processor generates the end-of-time-slice event.

The time-slice flag enables the generation of the end-of-time-slice event. Alternately, it can be used to prevent an end-of-time-slice event. If this bit is cleared, the processor will continue to execute the process beyond the expiration of its time-slice.

The time-slice-reschedule flag determines what the processor does when an end-of-time-slice event is generated. It can do either of two things: (1) generate a time-slice fault or (2) automatically suspend the current process and dispatch a new process. The latter function is one of the high-level process management actions discussed later in this chapter.

When an end-of-time-slice event is generated, the processor performs the following actions:

1. It copies the next-time-slice value into the residual-time-slice field, setting the count for the next time the process is worked on.

2. It updates the execution time by adding the next-time-slice value to the value in the execution-time field.
3. It checks the time-slice flag. If the flag is clear, the processor continues working on the process; if the flag is set, it goes to the next step.
4. It checks the time-slice-reschedule flag. If the flag is set, it automatically suspends the current process and dispatches a new process. If the flag is clear, it signals a time-slice fault and invokes the time-slice fault handler.

The time-slice fault action has two uses. One use is to allow the fault handler to alter process attributes such as the next-time-slice value or the priority before the process is suspended.

The other use of the time-slice fault is to support a kernel that is using the **saveprcs** and **resumprrcs** instructions to suspend and dispatch processes. Here, the fault handler can carry out process suspension and dispatching action.

### Execution-Time Counting

The execution-time field gives the elapsed execution time of the process. As shown in the above action statement, this field is only updated at the end of each time-slice, by adding the value in the next-time-slice field to the value in the execution-time field. (At the beginning of a time slice, the value in the execution-time field is thus equal to the actual elapsed time of the process plus the value of the next time slice.) The time that a process has spent in the execution state, at any given time, is then the value in the execution-time field minus the value in the residual-time-slice field.

The load-process-time instruction (**ldtime**) allows a process to determine its elapsed time during execution. This instruction stores the execution-time minus the residual-time-slice value in a specified register.

In a new PCB, the execution-time field should be set equal to the next-time-slice field.

The execution-time field can be used to count elapsed process-execution time even if time-slice scheduling is not used. To do this, the timing flag in the process controls must be set and the time-slice flag must be cleared. The processor then updates the execution-time field at the end of each time slice, but continues working on the process until the process is killed or blocked.

## OVERVIEW OF HIGH-LEVEL PROCESS MANAGEMENT FACILITIES

The processor's high-level process management facilities, introduced earlier in this chapter, allow the processor to handle the scheduling and dispatching of multiple processes automatically. The major benefits of these facilities are that:

- they provide a flexible and highly-efficient mechanism for managing processes in a multitasking system.
- they relieve a significant burden from the kernel.
- they simplify the design of multiple-processor systems.

The remainder of this chapter describes how these high-level facilities can be used for process scheduling and dispatching, process synchronization, and interprocess communication.

## Ports

The processor's high-level process-management facilities are based on ports. A port is a device for exchanging messages. It allows messages to be exchanged between two or more processes. A port can also be used to maintain a list of ready-to-execute processes for the processor.

A message is a segment that contains a queue record so that it can be queued to a port. A common type of message segment is a PCB that represents a ready-to-be-executed or blocked process. A message segment can also contain data that is to be exchanged between two processes. Messages are identified by their respective SS's.

A port is contained in a port segment. (The segment-descriptor format for port segments is given in Chapter 8.) As shown in Figure 14-2, the processor recognizes two types of ports: a first-in, first-out port (FIFO port) and a priority port. A FIFO port supports a single message queue; a priority port supports 32 message queues arranged in order of message priority.

Bit 16 of the first word of a port segment determines whether it is a FIFO port (clear) or a priority port (set).

## FIFO Port

A FIFO port contains a single, linked list (i.e., queue) of messages, arranged in FIFO order. When a message is received from a FIFO port, the message comes from the head (first message) of the queue.

The functions of the fields in a FIFO port are as follows. The *lock* field is used to synchronize the manipulation of a port by several processors. When a processor requires access to a port, it first checks bit 0 of the lock field. If this bit is 0, the processor atomically sets the bit to 1. The processor then accesses the port as needed. If the bit is 1 when the processor checks it, indicating that another processor is already accessing the port, the processor spins on the port, until the port becomes available for access.

Bit 17 of the first word of the port (the *queue-state* flag) shows what the port's queue is being used for. If this bit is set to 1, the queue contains blocked processes, waiting for messages; if the bit is set to 0, the queue contains messages waiting to be received or is empty.

The *queue-head SS* and *queue-tail SS* fields contains the SS of the message at the head of the queue and the tail of the queue, respectively. The messages in the queue are linked together through their respective queue records (described later in this chapter). A value of 0 in the queue-head-SS field indicates an empty queue.

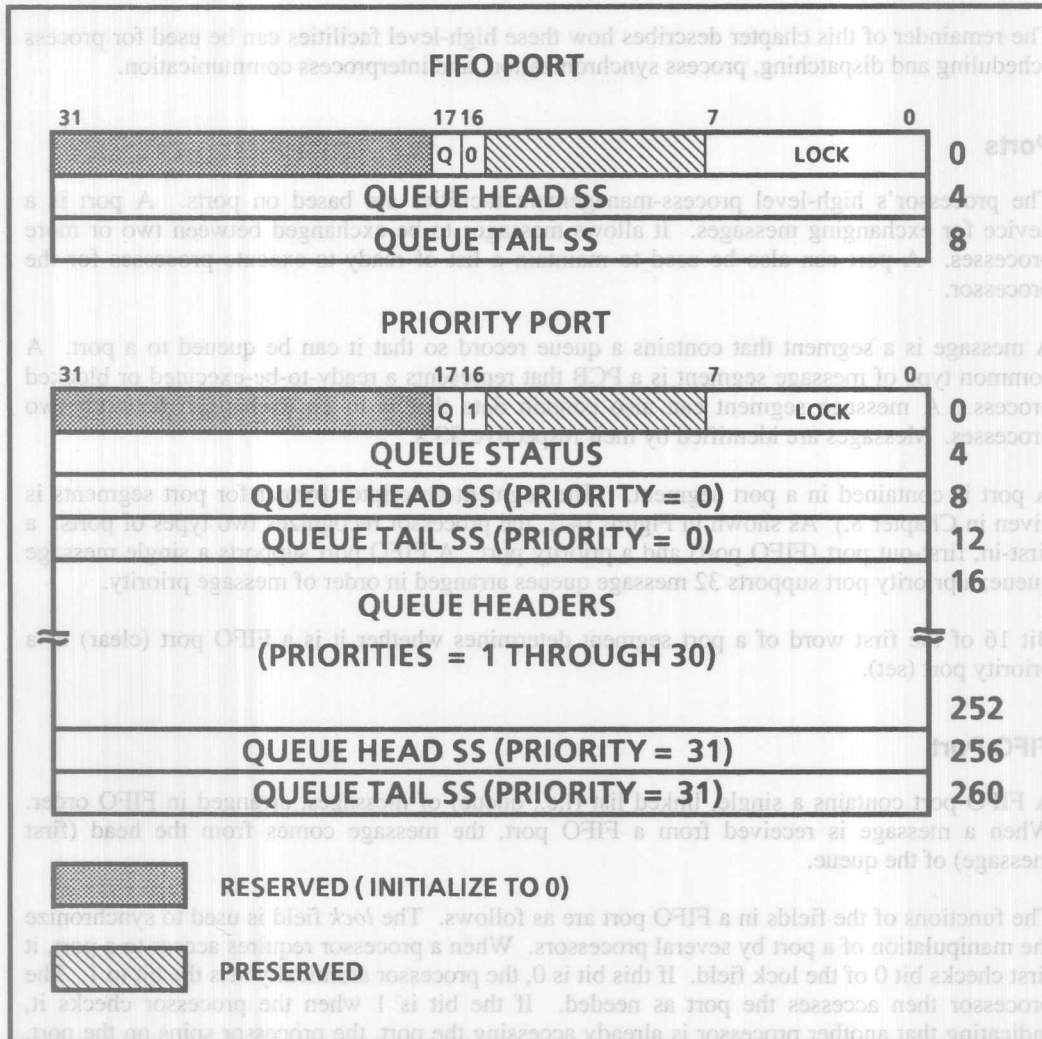


Figure 14-2: Ports

**Priority Port**

A priority port contains 32 queues (linked lists) of messages, with each queue arranged in FIFO order. When a message is received from a priority port, the message comes from the head of the highest priority non-empty queue. The priorities of the queues range from 0 to 31, with 31 being the highest priority.

The functions of the fields in a priority port are as follows. The *lock* field and the *queue-state* flag perform the same functions in the priority port as in the FIFO port.



The *queue-status* field shows the status of each queue in the port. Each bit in this field represents the state of one queue (with bit 0 representing the priority 0 queue, bit 1 the priority 1 queue, etc.). If a bit is set to 1, it indicates that the queue contains one or more messages; if the bit is set to 0, the queue is empty. If all the bits in the queue-status field are 0, the port is empty. Each of the 32 queues in the priority port is represented by a queue header. Each queue header is made up of a *queue-head-SS* field and a *queue-tail-SS* field. These fields perform the same functions as the corresponding fields in the FIFO port.

## Message

Any of the unpagged segments described in Chapter 8 can be used as a message segment, including a process segment, port segment, procedure-table segment, simple-region segment, and semaphore segment. When any of these segments is used as a message segment, the processor assumes that the first two words of the segment contain a queue record, shown in Figure 14-3.

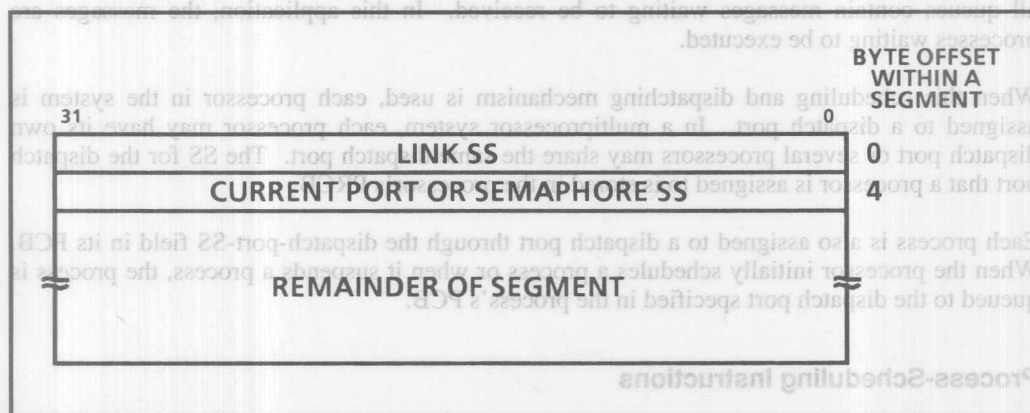


Figure 14-3: Queue Record

The queue record has two fields. The *link SS* field contains the SS of the message segment behind it in the queue. The *current-port SS* or *current-semaphore SS* field gives the port or semaphore that the message is queued to. The processor maintains the information in the queue record, independently from the software. When a message segment is created, the link SS and current-port-or-semaphore-SS fields should be set to 0.

Refer to the section later in this chapter titled "Interprocess Communication" for a discussion of how messages can be used for passing information between processes.

## Port Uses

The processor uses ports in two ways: as dispatch ports or as communication ports. A dispatch port is a device to assist the processor in scheduling processes. When the kernel creates a process, it queues it to the dispatch port for the processor. The processor then



dispatches processes from the dispatch port one at a time to work on them. When the processor suspends a process, it reschedules it at its dispatch port. The use of dispatch ports is described in the next section.

A communication port is used to pass messages between processes. These messages can be used to synchronize multiple processes in a multitasking environment or to share data among processes. The use of communication ports is discussed in the section later in this chapter titled "Interprocess Communication."

## AUTOMATIC PROCESS DISPATCHING

The priority-port data structure (described earlier in this chapter) provides a mechanism for the processor to maintain a list of scheduled processes, which it can then dispatch one at a time, independently from the kernel. Referring to Figure 14-1, when a priority port is used as a dispatch port, it contains a list of all the processes in the ready state.

A dispatch port must always be a priority port with its queue-state bit set to 0, indicating that all queues contain messages waiting to be received. In this application, the messages are processes waiting to be executed.

When this scheduling and dispatching mechanism is used, each processor in the system is assigned to a dispatch port. In a multiprocessor system, each processor may have its own dispatch port or several processors may share the same dispatch port. The SS for the dispatch port that a processor is assigned to is stored in the processor's PRCB.

Each process is also assigned to a dispatch port through the dispatch-port-SS field in its PCB. When the processor initially schedules a process or when it suspends a process, the process is queued to the dispatch port specified in the process's PCB.

### Process-Scheduling Instructions

The processor provides two instructions to support this automatic-dispatching mechanism. The schedule-process instruction (**schedprcs**) causes the processor to enqueue a process (i.e., its PCB) to a dispatch port. For example, if the kernel issues a **schedprcs** instruction for a process with a priority of 23, the process is placed at the front (or head) of the priority-23 queue of the dispatch port.

The send-service instruction (**sendserv**) causes the processor to suspend the process that it is currently executing and enqueue it at a specified port. This port may be the process's dispatch port or a communication port. If the port is a priority port, the processor checks the process's priority and places the process at the end (or tail) of queue for that priority.

### Process-Dispatching Action

The actions that the kernel and processor take to dispatch processes using the automatic-dispatching mechanism are as follows:

1. The kernel creates a process. In doing this, it allocates segments for regions 0, 1, and 2 of the process and for the process's PCB. It then creates an initial PCB for the process.
2. The kernel enqueues the process to the process's dispatch port, using the **schedprcs** instruction.
3. When the processor completes work on its current process, it suspends the process and reschedules it on the dispatch port.
4. The processor examines the dispatch port. If the port contains waiting processes, the processor goes to the highest priority, non-empty queue and dispatches the process from the head of this queue. The processor then binds itself to the process and begins executing it.
5. Upon completion of work on this process, the processor repeats steps 3 and 4 to reschedule the current process and dispatch another process.

Note that since the processor always goes to the highest, non-empty queue to dispatch a process, it will work on one priority queue alone until all the processes in that queue have been completed or killed (resulting in the processes being removed from the dispatch port); blocked at communication ports; or moved to lower priority queues.

### Process Suspension

Once the processor begins work on a process it will continue to work on it until it receives a signal to suspend the process. This signal can be caused by several events:

- End-of-time-slice event
- Process becomes receive or wait blocked
- Execution of a **sendserv** instruction
- Process becomes preempted by another higher-priority process

The end-of-time-slice event is used by the time-slice-scheduling mechanism described earlier in this chapter.

Process blocking is related to the semaphore and interprocess communication mechanisms described later in this chapter. A process can become blocked in either of two ways. One way is if the process attempts to receive a message from a communication port, but the message is not available. The processor then suspends the process and enqueues it on the communication port to await the message. The other way is if the process attempts to receive a signal from a semaphore, but none is available. Here, the processor suspends the process and enqueues it on the semaphore to await the signal.

A **sendserv** instruction can be executed from a fault or interrupt handler, or it can be included in the process code, if it is known beforehand that the process must always be suspended at a certain point.

Process preemption is described later in this chapter in the section titled "Process Preemption."

## PROCESS SYNCHRONIZATION

The process synchronization facilities of the processor allow the activities of several interacting processes to be synchronized. An important application of these facilities is to prevent race conditions between processes, particularly in multiprocessor systems, where two or more processes are being worked on simultaneously.

The processor provides two mechanisms that can be used to synchronize processes: semaphores and communication ports. Semaphores are described in this section. The use of communication ports for process synchronization are described later in this chapter in the section titled "Interprocess Communication."

### Use of Semaphores

A semaphore is a device for synchronizing the activities of several agents, in this case several processes. The following example shows one application of a semaphore.

Assume that process A and process B perform different but interdependent tasks and that at various points within the execution of process A, it must check that process B has completed execution of a particular task. To exchange information about the state of the task, the processes use a semaphore.

Each time process B completes the task, it increments a counter at the semaphore. Each time process A reaches a point in its execution where it needs to know if process B has completed the task or not, it checks the count at the semaphore. If the count has been incremented, process A decrements the count and continues executing. If the count has not been incremented (meaning that process B has not yet completed the task), the processor suspends process A and queues it to the semaphore. Process A is then said to be blocked at the semaphore.

When process B completes the task and goes to the semaphore, the processor sees process A queued to the semaphore. Then, instead of incrementing the count, the processor unblocks process A from the semaphore, freeing both process A and process B to continue execution.

### Semaphore Structure

A semaphore is contained in an embedded segment. (The segment-descriptor format for an embedded segment is given in Chapter 8.) The format for a semaphore is shown in Figure 14-4. The following paragraphs describe the fields of a semaphore.

The *lock* field of a semaphore performs the same function as the corresponding field of a port. It synchronizes the manipulation of a semaphore by several processors. When a processor needs to access a semaphore, it first checks bit 0 of the lock field. If this bit is 0, it atomically sets the bit to 1 and accesses the semaphore. If the bit is set to 1, the processor spins on the lock until the semaphore is available.

The *count* field contains a 16-bit ordinal. This field shows the number of times that one process has sent a signal to another process without the signal being received. The mechanism for incrementing and decrementing the count is described later in this chapter in the section titled "Semaphore-Access Actions."

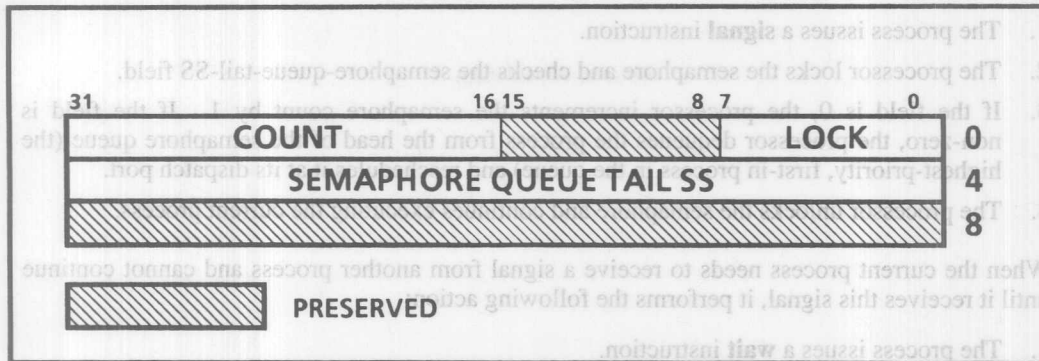


Figure 14-4: Semaphore Structure

The *semaphore-queue-tail-SS* field contains the SS of the last process in the semaphore queue. If no processes are queued to the semaphore, this field is set to 0.

The semaphore queue consists of a linked list of PCBs, with the linking carried out through their queue records. The processes in the queue are arranged in decreasing-priority order and FIFO within the same priority level. The link field in the queue record of the last process in the queue contains the SS of the first process in the queue.

### Semaphore-Handling Instructions

Three instructions are provided to handle communication with a semaphore. A process uses the **signal** instruction to send a signal to a semaphore that a task is complete. This instruction causes the processor to check the semaphore-queue-tail-SS field and either (1) increment the semaphore count, if the queue-tail value is 0, or (2) dequeue and reschedule the first process from the queue.

A process uses the **wait** and **condwait** (conditional-wait) instructions to receive a signal from a semaphore. The **wait** instruction causes the processor to check the semaphore count field and either (1) decrement the count, if it is non-zero, or (2) suspend the process and queue it to the semaphore, if the count is 0.

The **condwait** instruction performs a similar function, except that the process is not suspended and sent to the semaphore to wait if the count is 0. Instead, the processor either (1) decrements the count, if it is non-zero, or (2) does nothing to the semaphore, if the count is 0. In either case, the processor sets the condition code bits to indicate which action was taken.

### Semaphore-Access Actions

The actions that processes and the processor take to communicate through the semaphore mechanism are as follows.

When the current process needs to signal another process that a task is complete, it performs the following action:

1. The process issues a **signal** instruction.
2. The processor locks the semaphore and checks the semaphore-queue-tail-SS field.
3. If the field is 0, the processor increments the semaphore count by 1. If the field is non-zero, the processor dequeues the process from the head of the semaphore queue (the highest-priority, first-in process in the queue) and reschedules it at its dispatch port.
4. The processor unlocks the semaphore and continues executing the current process.

When the current process needs to receive a signal from another process and cannot continue until it receives this signal, it performs the following action:

1. The process issues a **wait** instruction.
2. The processor locks the semaphore and checks the semaphore count field.
3. If the count is non-zero, the processor decrements the semaphore count by 1, unlocks the semaphore, and continues executing the current process.
4. If the count is 0, the processor suspends the current process, queues it in the semaphore queue, unlocks the semaphore, and goes to the dispatch port to dispatch another process. Processes are enqueued in a semaphore queue in decreasing priority order and FIFO within a priority level.

When the current process needs to receive a signal from another process, but does not need to discontinue processing if the signal is not available, it performs the following action:

1. The process issues a **condwait** instruction.
2. The processor locks the semaphore and checks the semaphore count field.
3. If the count is non-zero, the processor decrements the semaphore count by 1. If the count is 0, the processor does nothing further to the semaphore.
4. The processor sets the condition code bits to 010<sub>2</sub> if the signal was received or to 000<sub>2</sub> if a signal was not received.
5. The processor unlocks the semaphore and continues executing the current process.

## PROCESS PREEMPTION

The processor provides a mechanism that allows a process (called a preempting process) to cause the processor to check the dispatch port and to suspend (or preempt) the current process if a higher priority process is found. To be a preempting process, the preempt flag in the process's process-controls word must be set. Preemption can occur in two situations: when a preempting process becomes unblocked from a semaphore or communications port; and when a processor receives a preemption IAC from another processor.

This preemption mechanism performs two functions. First, it provides a means of coordinating the activities of several processes that are performing cooperative tasks. Second, in multiprocessor systems, to help insure that the available processors are working on highest priority tasks.



The following paragraphs describe the preemption action that a single processor takes when a preempting process becomes unblocked; the section in Chapter 15 titled "Multiprocessor Preemption" describes preemption in multiprocessor systems.

### Process-Preemption Action

If a process becomes unblocked from a semaphore or a communication port and its preempt flag is set, the processor performs the following action. (Communication ports are described in the following section titled "Interprocess Communication".)

1. The processor enqueues the preempting process at the dispatch port.
2. If the current process is in the interrupted state, the processor sets the check-dispatch-port flag in the processor controls skips the remaining preemption actions. (When the processor returns from the interrupt, it checks the dispatch port automatically and dispatches the highest priority process that has a priority higher than the current process.)
3. The processor compares the priority of the preempting process with that of the current process.
4. If the priority of the preempting process is equal to or lower than that of the current process, the processor does not perform the preemption action.
5. If the priority of the preempting process is higher than that of the current process, the processor performs the remaining preemption steps.
6. The processor suspends the current process and places it on the dispatch port at the head of its priority queue.
7. The processor dispatches the highest priority process from the dispatch port and begins executing that process.

Two things should be noted about this mechanism. First, it is intended that the preempt flag be set for processes above a given priority level so that they will preempt lower priority processes immediately. Second, this mechanism does not insure that the preempting process is the next process dispatched, unless it is the highest priority process queued at the dispatch port.

### INTERPROCESS COMMUNICATION

The semaphore data structure, described in the previous section, provides a simple, efficient means of synchronizing the activity of several interacting processes. This section describes the use of the communication port data structure and messages in interprocess communication. As is shown in this section, a communication port is a more general data structure that not only supports process synchronization, but also can be used to pass messages and data structures between processes.

#### Communication Ports

A communication port is similar to a dispatch port except that it queues messages waiting to be received or processes waiting to receive messages. (A dispatch port queues processes waiting to be executed.) For example, if process A needs to send a message to process B, it sends the



message to a mutually agreed upon communication port. The processor then checks to see if process B is queued at the port, waiting for a message. If it is, the processor passes the message to process B and schedules process B at its dispatching port. If process B has a higher priority than process A, process A is preempted (suspended) and process B is dispatched. Otherwise, the processor resumes executing process A.

If process B is not waiting at the communication port when the message is sent, the processor queues the message on the port. Then when process B attempts to receive a message from the communication port, the processor takes the message from the message queue, passes it to process B, and continues executing process B.

A communication port can be either a FIFO port or a priority port (as shown in Figure 14-2). Bit 16 of the first word of the port data structure determines the port type. If the port is a FIFO port, messages or processes are queued to a single FIFO queue. If a port is a priority port, messages are queued to any of 32 queues according to their priority.

The Q bit (bit 17) of the first word determines whether a port contains blocked processes that are waiting to receive messages (the Q bit is set to 1) or whether it contains messages waiting to be received (the Q bit is set to 0). If the port is empty (it contains neither waiting processes or waiting messages) the Q bit is set to 0.

Thus, when a port is initially created, bit 16 of the first word should be set to 0 or 1, depending on the type of port being created, and bit 17 should be set to 0, indicating an empty port. Thereafter, the processor sets or clears the Q bit.

### Interprocess-Communication Mechanism

As with the self-dispatching mechanism of the processor, the processor handles the passing of message SS's back and forth among processors automatically and independently from the kernel. All the kernel is required to do is to set up the communication ports and create and fill the message segments.

To initiate the sending and receiving of messages, the processor provides four instructions: **send**, **receive**, **condrec** (conditional receive), and **sendserv** (send service). The processor must be in the supervisor mode to execute any of these instructions.

The following paragraphs summarize the actions that the processor performs for each of these instructions. Refer to the reference information on each instruction in Chapter 17 for more detailed descriptions of the instruction actions.

#### Send Message

The **send** instruction has three operands: communications port SS, message SS, and message priority. When a process issues a **send** instruction, the processor performs the following actions:

1. It checks the Q bit of the selected port. If the bit is set (indicating that processes are queued at the port waiting for messages), the processor finds the highest priority queue

- that contains waiting (blocked) processes and finds the first process from this queue. (If the port is a FIFO port, it finds the first process from the queue.)
2. It unblocks this process, loads the message SS into the message SS field of the process's PCB, and reschedules the process at its dispatching port.
  3. If the Q bit is clear (indicating an empty port or a port with waiting messages), the processor queues the message segment at the end of the queue specified with the message priority operand in the **send** instruction. (If the port is a FIFO port, the message priority is ignored.)
  4. Following either of the above actions, the processor resumes execution of the current process (the process that sent the message).

### Receive a Message

The two receive instructions (**receive** and **condrec**) allow a process to pick up a message SS from a communication port. The **receive** instruction has two operands: a port SS and a destination register where the receive message is to be stored. When a process issues a **receive** instruction, the processor performs the following actions:

1. It checks the Q bit of the selected port. If the bit is clear (indicating a port with waiting messages or an empty port), the processor finds the highest priority queue that contains queued messages. (If the port is a FIFO port, it looks only at the single FIFO queue.)
2. The processor then takes the first message from the queue, stores it in the destination register specified in the **receive** instruction, and resumes execution of the process.
3. If the port is empty (all queues are empty) or has waiting processes (Q bit is set), the processor suspends the current process (receiving process) and queues it at the end of the queue specified with the priority field in the process's process controls word. (If the port is a FIFO port, the process priority is ignored.)
4. The processor then dispatches another process from the dispatch port and begins executing that process.

With the **condrec** instruction, the processor performs the same operation as it does with the **receive** instruction, except that it does not block the process at the communication port if there is no message available. Instead it sets the condition code bits in the arithmetic controls to indicate that a message was not received and resumes execution of the receiving process.

The **condrec** instruction has the same operands as the **receive** instruction: port SS and a destination register where the receive message is to be stored. When a process issues a **condrec** instruction, the processor performs the following actions:

1. It checks the Q bit of the selected port. If the bit is clear (indicating that there is a port with waiting messages or the port is empty), the processor finds the highest priority queue that contains queued messages. (If the port is a FIFO port, it looks only at the single FIFO queue.)
2. The processor then takes the first message from the queue, stores it in the destination register specified in the **condrec** instruction, sets the condition code to 010<sub>2</sub>, and resumes execution of the process.

3. If the port is empty (all queues are empty) or has waiting processes (Q bit is set), the processor sets the condition code to 000<sub>2</sub> and resumes execution of the process.

If a process fails to receive a message after issuing a **condrec** instruction, one action that the process can take is to wait for several ticks, then issue a **condrec** instruction again.

### Send Service

The **sendserv** instruction offers a special application of the message passing mechanism. This instruction causes the processor to suspend the current process and send its SS as a message to a communication port.

This instruction has one operand, the SS of the communication port to receive the suspended process's SS. When the **sendserv** instruction is issued, the processor performs the following action:

1. It suspends the current process and goes to the communication port specified in the port SS operand of the instruction.
2. It checks the Q bit of the selected port. If the bit is set (indicating that processes are queued at the port), the processor finds the highest priority queue that contains waiting processes and finds the first process from this queue. (If the port is a FIFO port, it finds the first process from the queue.)
3. It unblocks this process, loads the SS for the suspended process into the message SS field of the waiting process's PCB, and reschedules the waiting process at its dispatching port.
4. If the Q bit is clear (indicating an empty port or a port with waiting messages), the processor queues the suspended process's PCB at the end of the queue specified with the process's priority field in its process controls. (If the port is a FIFO port, the process priority is ignored.)
5. Following either of the above actions, the processor dispatches a new process from the dispatching port and begins executing it.

The use of this instruction is described in the section later in this chapter titled "Applications of Messages."

### Kernel Support for Message Passing

In general, the kernel must provide some support code to make interprocess-communication services available to application programs. Typically, kernel procedures are written that allow an application program to send and receive messages by making system calls (**calls** instruction) to the kernel.

These kernel procedures take care of setting up communication ports and creating message segments. Then, to send a message to another process, all that an application program has to do is supply a data word or a pointer to a data structure as a parameter in a system call to a send procedure. The kernel procedure will then load the word, the pointer, or a whole data structure into a message segment and issue a **send** instruction to send the message segment to a communication port.

Likewise, to receive a message from another process, an application program issues a system call to a receive procedure. The kernel then issues a **receive** or **condrec** instruction, gets the message SS, retrieves the data word, pointer, or data structure from the message segment, and returns it to the application program as a parameter.

### Applications of Messages

The message passing mechanism can be used in several ways for either process synchronization or the passing of information between processes.

One application of a communication port is to synchronize processes in a manner similar to that described for a semaphore. Here, instead of incrementing a counter as a signal from one process to another, a message segment is left at a communication port.

The message segment can be used in two ways. First, it can contain a null message, in which case the passing of message SS's would be used strictly to synchronize processes. Second, the message segment can be encoded to contain information about the respective processes.

One of the benefits of using communication ports instead of semaphores for process synchronization is that processes waiting for messages can be prioritized.

When messages are used to pass information between processes, the message segments are typically mapped into predefined areas of region 3, and cooperating processes know the conventions of this mapping. One process can then pass data to another process by writing the data into a predefined message area and sending a pointer to that area to the kernel. The kernel then handles the message passing and returns the pointer to the receiving process.

The message being passed can also be a processor-defined data structure such as a port or a PCB. For example, a kernel may create communication ports dynamically. It could then send a new port as an SS to a process for use in the future for sending and receiving messages.

The **sendserv** instruction as is described above is specifically designed to send PCBs as messages. This instruction allows a process to explicitly suspend itself at a specific point in its activity. This capability has two common applications. One is to allow the process to suspend itself and have the processor reschedule it at a dispatch port. The other is to allow the processor to automatically kill processes that have completed their tasks. Here, the **sendserv** instruction sends the process's PCB to a communication port set up to handle dead processes. Another process then periodically takes the PCB messages from this communication port, deallocates the system resources that have been allocated to them, and deletes or frees up the PCBs.

Likewise, to receive a message from another process, an application program issues a system call to a receive procedure. The kernel then issues a receive or conduct instruction, gets the message SS, retrieves the data word, pointer, or data structure from the message segment, and returns it to the application program as a parameter.

## Applications of Messages

The message passing mechanism can be used in several ways for either process synchronization or the passing of information between processes.

One application of a communication port is to synchronize processes in a manner similar to that described for a semaphore. Here, instead of incrementing a counter as a signal from one process to another, a message segment is left at a communication port.

The message segment can be used in two ways. First, it can contain a null message, in which case the passing of message SS's would be used strictly to synchronize processes. Second, the message segment can be encoded to contain information about the respective processes.

One of the benefits of using communication ports instead of semaphores for process synchronization is that processes waiting for messages can be prioritized.

When messages are used to pass information between processes, the message segments are typically mapped into predefined areas of region 3, and cooperating processes know the conventions of this mapping. One process can then pass data to another process by writing the data into a predefined message area and sending a pointer to that area to the kernel. The kernel then handles the message passing and returns the pointer to the receiving process.

The message being passed can also be a processor-defined data structure such as a port or a PCB. For example, a kernel may create communication ports dynamically. It could then send a new port as an SS to a process for use in the future for sending and receiving messages.

The senderv instruction as is described above is specifically designed to send PCBs as messages. This instruction allows a process to explicitly suspend itself at a specific point in its activity. This capability has two common applications. One is to allow the process to suspend itself and have the processor reschedule it at a dispatch port. The other is to allow the processor to automatically kill processes that have completed their tasks. Here, the senderv instruction sends the process's PCB to a communication port set up to handle dead processes. Another process then periodically takes the PCB messages from this communication port, deallocates the system resources that have been allocated to them, and deletes or frees up the PCBs.

---

# *Multiple-Processor Operation*

---

**15**





## CHAPTER 15

# MULTIPLE-PROCESSOR OPERATION

This chapter presents several features of the processor that support multiprocessor systems. Included are discussions of external IAC messages, high-level process management facilities, atomic instructions, and interrupt handling.

### OVERVIEW OF MULTIPROCESSOR-SUPPORT FACILITIES

The processor provides several facilities that greatly simplify the design of systems that use multiple processors, particularly in applications in which the processors share memory and processing tasks. These facilities include the following items:

- External IAC messages
- High-level process management facilities
- Atomic instructions
- Shared interrupt-handling facilities

### EXTERNAL IAC MESSAGES

Chapter 11 presents the concept of an interagent communication (IAC) and describes how internal IACs are sent. (An internal IAC is one that a processor sends to itself). This section describes how external IACs are sent from one processor to another.

External IACs are used by agents external to the processor to initiate processor actions such as testing for pending interrupts or freezing the processor. External IACs can be sent between two 80960MC processors that are connected to the same bus or by external logic that duplicates the external IAC sending mechanism. The following sections describe how one processor sends an IAC to another processor. The *80960MC Hardware Designer's Reference Manual* describes the requirements that external logic must meet to perform these same functions.

#### Sending External IACs

Sending an external IAC message is similar to sending an internal IAC message, except that the address of the receiving processor is specified in a slightly different way. (Internal IACs are always sent to address FF000010<sub>16</sub>.)

The IAC message format is the same as is shown in Figure 11-1. Figure 15-1 shows how the address for the receiving processor is encoded.

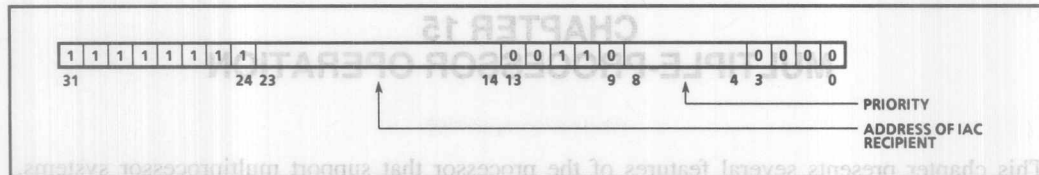


Figure 15-1: Encoding of Address for Processor Receiving an IAC

At initialization each processor is assigned a unique physical address in the range of  $FF000C00_{16}$  to  $FFFFCC00_{16}$ . To send an IAC to a processor, the sending processor sends the message to the physical address assigned to the receiving processor. As shown in Figure 15-1, only bits 14 through 23 of this address are interpreted to determine the address of the receiving processor. Bits 4 through 8 of this address are used to encode the priority of the message.

For example, to send a priority  $25_{10}$  IAC to the processor at address  $0000000001_2$ , the message physical address would be  $FF004D90_{16}$ .

#### NOTE

If virtual addressing is being used, the address accompanying the IAC must be mapped to the physical address assigned to the receiving processor.

To send an external IAC, software must perform the following steps:

1. Load the message into four consecutive words in memory, with the first word aligned on a word boundary.
2. Execute a **symovq** instruction to move the message from its source address to the address of the receiving processor (encoded in the form shown in Figure 15-1).
3. Check the condition code in the arithmetic controls to determine if the message was received ( $010_2$ ) or rejected ( $000_2$ ).

The action of the **symovq** move instruction insures that the sending processor does not execute any other instructions until the **symovq** instruction is complete. It also sets the condition code bits to indicate whether or not the move was successful. A successful move is interpreted as the IAC being received by the processor. As is discussed in the next section, external logic may be employed to intercept IACs and reject them if their priorities (as encoded in the message address) are equal to or less than the task the processor is currently working on. The process running on the sending processor then has the option of sending the IAC again at a higher priority or sending the IAC repeatedly at the same priority until it is accepted.

#### Receiving and Handling External IACs

A processor receives and handles an external IAC in somewhat the same manner as it receives and handles an interrupt. To configure a processor to receive external IACs, vector **INT0** of the interrupt-control register (shown in Figure 10-2) is set to 0. The **INT0** pin on the processor chip then becomes the IAC pin. (Refer to the section in Chapter 10 titled "Interrupts From Interrupt Pins" for further discussion of the interrupt pins and interrupt-control register.)

When the processor receives a signal on the  $\overline{\text{IAC}}$  pin, it handles it initially as if it were receiving an interrupt. It reads the vector number associated with this pin (bits 0 through 7 of the interrupt-control register). If it is zero, the processor recognizes that it is receiving an external IAC. It then reads the four-word IAC message from the local bus and performs the requested IAC.

Since the processor handles IACs with a mechanism that is separate from the process-execution mechanism, it does not save the state of the current process prior to handling an IAC. Once a processor has finished handling an IAC, it resumes work on the current process, unless the action specified with the IAC (such as a processor restart or a process preemption) makes this impossible.

The processor acts immediately on any IAC that it receives. For efficient system operation, external logic must thus be provided to insure that low priority IAC messages do not interrupt the processor while it is handling a higher priority task. This logic is usually supplied by the M82965 component.

To support the M82965 (or other external logic) in this job, the processor provides a mechanism, called the write-external-priority mechanism, which periodically writes the priority of the processor's current task out on the bus as an IAC message. (The write-external-priority flag in the processor controls word enables this mechanism, as described in Chapter 9). The M82965 reads this message and keeps track of the current priority of the processor.

When an IAC is sent to the processor, the M82965 intercepts it and reads the priority encoded in the IAC address. It then determines whether the IAC priority is above that of the process currently running on the processor or not. If the IAC has a higher priority, the M82965 sends an acknowledge signal to the sending processor, then signals the receiving processor by asserting its  $\overline{\text{IAC}}$  pin. If the IAC has an equal or lower priority, the M82965 sends a not-acknowledged signal to the sending processor.

The sending processor uses the acknowledge or not-acknowledged signals to set the condition codes to complete the **synmovq** instruction.

While a processor is servicing an IAC, it performs some handshaking with its M82965 so that the M82965 knows when the processor has finished work on an IAC. The M82965 is then able to reject any IAC that it receives while the processor is servicing another IAC.

Refer to the *80960MC Hardware Designer's Reference Manual* for further information on how the M82965 handles IAC messages.

## HIGH-LEVEL PROCESS MANAGEMENT FACILITIES

All of the process-management facilities are available for use in multiprocessor systems. Of these, two are of particular importance: process scheduling and dispatching, and process preemption.

The following fields in the PRCB and PCB control the multiprocessor-preemption mechanism: the multiprocessor-preempt flag, nonpreempt-limit field, interim-priority field, and write-external-priority flag in the processor controls; the multiprocessor-preemption field of the PRCB; and the preempt flag of the process controls.

## Process Scheduling and Dispatching

The processor's high-level process management facilities are particularly useful for scheduling and dispatching processes in a multiprocessor system. They provide an efficient method of distributing the processor resources among the tasks to be handled by the system. They also remove a significant burden from the kernel for handling process management.

How these facilities are used centers around how the dispatch port is used. If the intent of the system is to share the processing tasks evenly among the available processors, the system can use a single dispatch port that is shared by all the processors. All processes are thus scheduled and dispatched from the same place. The lock on the dispatch port allows processors to take turns dispatching and enqueueing processes from the port.

An alternate use of a dispatch port is to give each processor in the system its own port. The kernel is then responsible for determining the load on each processor, which it does by scheduling the ready processes on selected dispatch ports.

## Multiprocessor Preemption

When using the high-level process management facilities described in Chapter 14, the processor provides the ability for a higher priority process to preempt a lower priority process. This means that the processor suspends the current lower-priority process and dispatches the higher priority, preempting process. A process can be a preempting process only if the preempt flag in its process controls is set.

Typically, preemption happens when a process becomes unblocked from a semaphore after receiving a signal or from a communication port after receiving a message. If the unblocked process has a higher priority than the current process, the processor preempts the current process.

Often the preempted process is also a preempting process. If there are other processors in the system, the multiprocessor-preempt mechanism provides a means for the processor that suspended a preempting process to check if one of the processors in the system can handle the process. It does this by sending a preempt process IAC message to one or two other processors, as described in the next section. From the priority of the message, the receiving processor determines whether the priority of the preempting process is higher than the process it is currently working on. If it is, the receiving processor suspends its current process and dispatches the higher priority process from the dispatch port. If both of the processors are working on higher priority processes, the sending processor begins work on its current process.

This technique insures that if there are  $n$  processors in the system, the  $n$  highest-priority processes are always being run.

## Preemption Control

The following fields in the PRCB and PCB control the multiprocessor-preemption mechanism: the multiprocessor-preempt flag, nonpreempt-limit field, interim-priority field, and write-external-priority flag in the processor controls; the multiprocessor-preemption field of the PRCB; and the preempt flag of the process controls.



The multiprocessor-preempt flag enables the multiprocessor-preemption mechanism. When this flag is set, the processor carries out the multiprocessor-preemption actions automatically, with no intervention from the kernel required.

In carrying out the multiprocessor-preemption action, the processor sends preempt process IAC messages (85000000<sub>16</sub>) to one or two other processors in the system. The IAC message and the addresses of the IAC message buffers for the two processors are contained in the multiprocessor-preemption field of the PRCB. The addresses are placed in the first two words of the field and the preempt process IAC message is placed in the third word. The addresses are stored in the form shown in Figure 15-1. The priority encoded in the address word is generally chosen to be a low value. For example, if the priority is set to 1, only idle processors (those with a 0 priority) will accept the IAC message. Any IAC message can be stored in the message word, but preempt process IAC is used for multiprocessor-preemption applications.

The nonpreempt-limit field contains a threshold priority that a processor uses to determine whether or not to perform a preemption action when it receives a preemption IAC message. If the priority of the preempting process (as contained in the IAC message) is equal to or lower than the priority of the processor's current process or the nonpreempt limit, the processor rejects the IAC and continues work on its current process. Typically, the nonpreempt-limit field is set to the middle of the priority range (12 to 10) to prevent a processor from carrying out process switches to service low-priority preempting processes.

The write-external-priority flag controls whether or not the priority of the currently running process is written out on the processor's bus. When this bit is set, the current priority is written out to the bus (in the form of an IAC message) whenever the following things occur: a process switch, an interrupt not caused by an IAC message, the execution of a **modpc** instruction (modify process controls).

The purpose of the write-external-priority mechanism is to keep external agents on the bus apprised of the priority of the task the processor is currently performing. The agent can then block IAC messages that are of lower priority. For example, if M82965s are being used in the system, the M82965 associated with each processor keeps track of the processor's priority by means of write-external-priority messages from the processor. When one processor sends a preempt process IAC message to another processor, the M82965 for the receiving processor checks the priority of the message and rejects it if it is not higher than the current priority of the processor.

The interim-priority field of the processor controls provides a means of setting the processor's priority to a high-enough level to avoid being interrupted by IACs. This field is only used when the write-external-priority function is enabled. When this function is enabled, the processor writes the value in the interim-priority field out on the bus any time one of the following instructions are executed: **send**, **sendserv**, **signal**, and **schedprcs**. This field is typically set to a high priority value (25 to 30) to insure that these instructions are able to be completed before the processor is forced to service an IAC message.



### Multiprocessor-Preemption Action

The processor performs the following actions when the multiprocessor-preempt flag is set and the processor schedules a preempting process at the dispatch port:

1. The processor sends a preempt process IAC message from the multiprocessor-preemption field of the PRCB to the first address given in this field.
2. The M82965 associated with the receiving processor compares the priority of the IAC with the processor's current priority. If IAC priority is higher, the M82965 sends the IAC on to the receiving processor and sends an ACK signal back to the sending processor.
3. If the receiving processor is not interrupted or stopped, it checks the dispatch port to determine the priority of the highest-priority process queued at the port. It then compares this priority with that of its current process and its nonpreempt-limit field. If the priority of the process at the dispatch port is higher than that of either the current process or the nonpreempt limit, the receiving processor suspends its current process and dispatches the higher-priority process from the dispatch port. If the priority is lower, the receiving process resumes work on its current process.
4. Upon receiving the ACK signal from the M82965 for the receiving processor, the sending processor then resumes work on its current process.
5. If in step 2 the priority of the first receiving processor is higher than the IAC priority, the M82965 sends a NACK signal back to the sending processor indicating that it has rejected the IAC message.
6. The sending processor then sends the message to the next address in the multiprocessor-preemption field of the PRCB.
7. If the second receiving processor also rejects the IAC message, the sending processor sends an IAC back to the first receiving processor, but this time it sets the priority of the message equal to that of the preempting process.
8. Again, if this message is rejected, the sending processor sends the higher priority message to the second receiving processor.
9. If the IAC is rejected at both priorities by both receiving processors, the sending processor abandons its attempt to find a processor to preempt and resumes work on its current process.
10. If in process suspended in step 3 is also a preempting process, the receiving processor then performs this multiprocessor-preemption action to attempt to get either of two processors to work on the process.
11. This action is continued until the available processors are servicing the highest-priority processes.

### ATOMIC INSTRUCTIONS

The atomic instructions allow a processor in a multiprocessor system to perform certain read-modify-write operations on a memory location, with the guarantee that the write will be completed before another processor is allowed access to the memory location. This capability is essential for performing operations on certain data structures, where it is important that one processor does not alter the data structure while another processor is trying to perform a read-modify-write on it.

The processor provides two atomic instructions: atomic add (**atadd**) and atomic modify (**atmod**). The **atadd** instruction adds a 32-bit ordinal value to a 32-bit target value in memory. The **atmod** instruction inserts a 32-bit value into a memory location, under the control of a mask. The mask determines which of the target bits in memory are actually modified.

## INTERRUPT HANDLING IN A MULTIPROCESSOR SYSTEM

A useful feature of the interrupt table in a multiprocessor system is that it allows the handling of interrupts to be shared. In a multiprocessor system, each processor has its own interrupt stack, but all the processors can share the interrupt table.

If a processor receives an interrupt that is at an equal or lower priority than the process that it is currently working on, it posts the interrupt as a pending interrupt in the interrupt table. All the processors check for pending interrupts at certain times as described in Chapter 10 in the section titled "Pending Interrupts." If one processor is not able to handle an interrupt, another one is likely to be available.

The test-pending-interrupts IAC provides a means for one processor to explicitly request that another processor check for pending interrupts and handle them if they exist.

The processor provides two atomic instructions: atomic add (stadd) and atomic modify (atmod). The stadd instruction adds a 32-bit ordinal value to a 32-bit target value in memory. The atmod instruction inserts a 32-bit value into a memory location, under the control of a mask. The mask determines which of the target bits in memory are actually modified.

## INTERRUPT HANDLING IN A MULTIPROCESSOR SYSTEM

A useful feature of the interrupt table in a multiprocessor system is that it allows the handling of interrupts to be shared. In a multiprocessor system, each processor has its own interrupt stack, but all the processors can share the interrupt table.

If a processor receives an interrupt that is at an equal or lower priority than the process that is currently working on, it posts the interrupt as a pending interrupt in the interrupt table. All the processors check for pending interrupts at certain times as described in Chapter 10 in the section titled "Pending Interrupts." If one processor is not able to handle an interrupt, another one is likely to be available.

The test-pending-interrupts IAC provides a means for one processor to explicitly request that another processor check for pending interrupts and handle them if they exist.





## CHAPTER 16 DEBUGGING

This chapter describes the tracing facilities of the 80960MC processor, which allow the monitoring of instruction execution.

### OVERVIEW OF THE TRACE-CONTROL FACILITIES

The 80960MC processor provides facilities for monitoring the activity of the processor by means of trace events. A trace event in the 80960MC is a condition where the processor has just completed executing a particular instruction or type of instruction, or where the processor is about to execute a particular instruction.

By monitoring trace events, debugging software is able to display or analyze the activity of the processor or of a program. This analysis can be used to locate software or hardware bugs or for general system monitoring during the development of system or applications programs.

The typical way to use this tracing capability is to set the processor to detect certain trace events either by means of the trace-controls word or a set of breakpoint registers. An alternate method of creating a trace event is with the **mark** and force mark (**fmark**) instructions. These instructions cause an explicit trace event to be generated when the processor detects them in the instruction stream.

If tracing is enabled, the processor signals a trace fault when it detects a trace event. The fault handler for trace faults can then call the debugging monitor software to display or analyze the state of the processor when the trace event occurred.

### REQUIRED SOFTWARE SUPPORT FOR TRACING

To use the processor's tracing facilities, software must provide trace-fault handling procedures, perhaps interfaced with a debugging monitor. Software must also manipulate several control flags to enable the various tracing modes and to enable or disable tracing in general. These control flags are located in the system-data structures described in the next section.

### TRACE CONTROLS

The following flags or fields control tracing:

- Trace controls
- Trace-enable flag in the process controls
- Trace-fault-pending flag in the process controls
- Trace flag (bit 0) in the return-status field of register r0
- Trace-control flag in the supervisor-stack-pointer field of the system table or a procedure table



## Trace-Controls Word

The trace-controls word is located in the PCB for the current process. When a process is bound to the processor, the contents of the trace-controls word are cached internally in the processor.

The trace controls allow software to define the conditions under which trace events are generated. Figure 16-1 shows the structure of the trace-controls word.

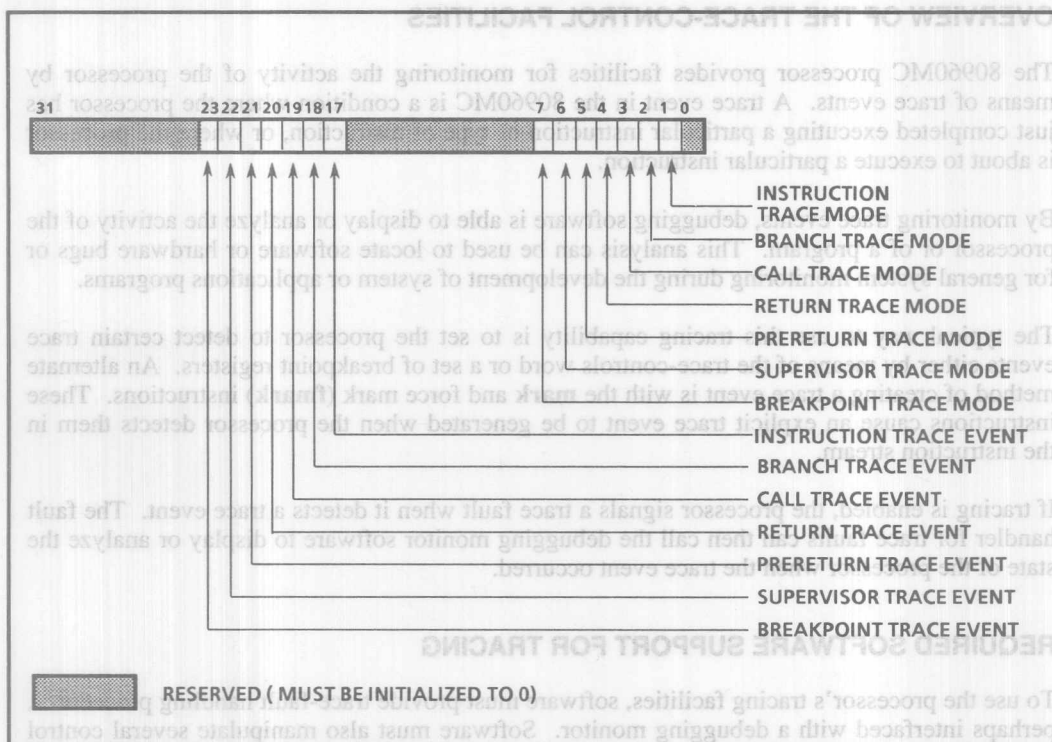


Figure 16-1: Trace-Controls Word

This word contains two sets of bits: the mode flags and the event flags. The mode flags define a set of trace modes that the processor can use to generate trace events. A mode represents a subset of instructions that will cause trace events to be generated. For example, when the call-trace mode is enabled, the processor generates a trace event whenever a call or branch-and-link operation is executed. To enable a trace mode, the kernel sets the mode flag for the selected trace mode in the trace controls. The trace modes are described later in this chapter.

The processor uses the event flags to keep track of which trace events (for those trace modes that have been enabled) have been detected.

A special instruction, the modify-trace-controls (**modtc**) instruction, allows software to set or clear flags in the trace controls. On initialization, all the flags in the processor's internal trace controls are cleared. The **modtc** instruction can then be used to set or clear trace mode flags as

required. (This instruction does not affect the trace controls word in the PCB for the current process.)

Software can access the event flags using the **modtc** instruction; however, there is no reason to. The processor modifies these flags as part of its trace-handling mechanism.

Bits 0, 8 through 16, and 24 through 31 of the trace controls are reserved. Software should initialize these bits to zero and not access or modify them thereafter.

### Trace-Enable and Trace-Fault-Pending Flags

The trace-enable flag and the trace-fault-pending flag, located in the process controls (shown in Figure 13-2), control tracing. The trace-enable flag enables the processor's tracing facilities. When this flag is set, the processor generates trace faults on all trace events.

Typically, software selects the trace modes to be used through the trace controls. It then sets the trace-enable flag when tracing is to begin. This flag is also altered as part of some of the call and return operations that the processor carries out, as described at the end of this chapter.

The trace-fault-pending flag allows the processor to keep track of the fact that an enabled trace event has been detected. The processor uses this flag as follows. When the processor detects an enabled trace event, it sets this flag. Before executing an instruction, the processor checks this flag. If the flag is set, it signals a trace fault. By providing a means of recording the occurrence of a trace event, the trace-fault-pending flag allows the processor to service an interrupt or handle a fault other than a trace fault, before handling the trace fault. Software should not modify this flag.

### Trace Control on Supervisor Calls

The trace flag and the trace-control flag allow tracing to be enabled or disabled when a call-system instruction (**calls**) is executed that results in a switch to supervisor mode. This action occurs independent of whether or not tracing is enabled prior to the call.

When a supervisor call is executed (**calls** instruction that references an entry in a procedure table with an entry type 11<sub>2</sub>), the processor saves the current state of the trace-enable flag (from the process controls) in the trace flag (bit 0) of the return-status field of register r0.

Then, when the processor selects the supervisor procedure from the procedure table, it sets the trace-enable flag in the process controls according to the setting in the trace-control flag in the procedure table (bit 0 of the word that contains the supervisor-stack pointer). When the trace-control flag is set, tracing is enabled; when it is clear, tracing is disabled.

On a return from the supervisor procedure, the trace-enable flag in the process controls is restored to the value saved in the return-status field of register r0.

## TRACE MODES

The following trace modes can be enabled through the trace controls:

- Instruction trace
- Branch trace
- Call trace
- Return trace
- Prereturn trace
- Supervisor trace
- Breakpoint trace

These modes can be enabled individually or several modes can be enabled at once. Some of these modes overlap, such as the call-trace mode and the supervisor-trace mode. The section later in this chapter titled "Handling Multiple Trace Events" describes what the processor does when multiple trace events occur.

The following sections describe each of the trace modes.

### Instruction Trace

When the instruction-trace mode is enabled, the processor generates an instruction-trace event each time an instruction is executed. This mode can be used within a debugging monitor to single-step the processor.

### Branch Trace

When the branch-trace mode is enabled, the processor generates a branch-trace event any time a branch instruction that branches is executed. A branch-trace event is not generated for conditional-branch instructions that do not branch. Also, branch-and-link, call, and return instructions do not cause branch-trace events to be generated.

### Call Trace

When the call-trace mode is enabled, the processor generates a call-trace event any time a call instruction (**call**, **callx**, or **calls**) or a branch-and-link instruction (**bal** or **balx**) is executed. An implicit call, such as the action used to invoke a fault handler or an interrupt handler, also causes a call-trace event to be generated.

When the processor detects a call-trace event, it also sets the prereturn-trace flag (bit 3 of register r0) in the new frame created by the call operation or in the current frame if a branch-and-link operation was performed. The processor uses this flag to determine whether or not to signal a prereturn-trace event on a **ret** instruction.

## Return Trace

When the return-trace mode is enabled, the processor generates a return-trace event any time a **ret** instruction is executed.

## Prereturn Trace

The prereturn-trace mode causes the processor to generate a prereturn-trace event prior to the execution of any **ret** instruction, providing the prereturn-trace flag in **r0** is set. (Prereturn tracing cannot be used without enabling call tracing.)

The processor sets the prereturn-trace flag whenever it detects a call-trace event (as described above for the call-trace mode). This flag performs a prereturn-trace-pending function. If another trace event occurs at the same time as the prereturn-trace event, the prereturn-trace flag allows the processor to fault on the non-prereturn-trace event first, then come back and fault again on the prereturn-trace event. The prereturn trace is the only trace event that can cause two successive trace faults to be generated between instruction boundaries.

## Supervisor Trace

When the supervisor-trace mode is enabled, the processor generates a supervisor-trace event any time (1) a call-system instruction (**calls**) is executed, where the procedure table entry is a supervisor procedure, or (2) when a **ret** instruction is executed and the return-status field is set to  $010_2$  or  $011_2$  (i.e., return from supervisor mode).

This trace mode allows a debugging program to determine the boundaries of operating-system calls within the instruction stream.

## Breakpoint Trace

The breakpoint-trace mode allows trace events to be generated at places other than those specified with the other trace modes. This mode is used in conjunction with the **mark** and force-mark (**fmark**) instructions, and the breakpoint registers.

The **mark** and **fmark** instructions allow breakpoint-trace events to be generated at specific points in the instruction stream. When the breakpoint-trace mode is enabled, the processor generates a breakpoint-trace event any time it encounters a **mark** instruction. The **fmark** causes the processor to generate a breakpoint-trace event regardless of whether the breakpoint-trace mode is enabled or not.

The processor has two, one-word breakpoint registers, designated as breakpoint 0 and breakpoint 1. Using the set-breakpoint-register IAC, one instruction pointer can be loaded into each register. The processor then generates a breakpoint trace any time it executes an instruction referenced in a breakpoint register.

## TRACE-FAULT HANDLER

A fault handler is a procedure that the processor calls to handle faults that occur. The requirements for fault handlers are given in Chapter 12 in the section titled "Fault-Handler Procedures."

A trace-fault handler has one additional restriction. It must be called with an implicit supervisor call, and the trace-control flag in the procedure-table entry must be clear. This restriction insures that tracing is turned off when a trace fault is being handled, which is necessary to prevent an endless loop.

## SIGNALING A TRACE EVENT

To summarize the information presented in the previous sections, the processor signals a trace event when it detects any of the following conditions:

- An instruction included in a trace-mode group is executed or about to be executed (in the case of a prereturn trace event) and the trace mode for that instruction is enabled.
- An implicit call operation has been executed and the call-trace mode is enabled.
- A **mark** instruction has been executed and the breakpoint-trace mode is enabled.
- An **fmark** instruction has been executed.
- An instruction specified in a breakpoint register is executed and the breakpoint-trace mode is enabled.

When the processor detects a trace event and the trace-enable flag in the process controls is set, the processor performs the following action:

1. The processor sets the appropriate trace-event flag in the trace controls. If a trace event meets the conditions of more than one of the enabled trace modes, a trace-event flag is set for each trace mode condition that is met.
2. The processor sets the trace-fault-pending flag in the process controls.

### NOTE

The processor may set a trace-event flag and the trace-fault-pending flag before it has completed execution of the instruction that caused the event. However, the processor only handles trace events in between the execution of instructions. When the processor encounters a **mark** instruction, it generates a breakpoint-trace event any time it encounters a **mark** instruction. The **mark** instruction generates a breakpoint-trace event any time it encounters a **mark** instruction. The **mark** instruction generates a breakpoint-trace event any time it encounters a **mark** instruction. The **mark** instruction generates a breakpoint-trace event any time it encounters a **mark** instruction.

## HANDLING MULTIPLE TRACE EVENTS

If the processor detects multiple trace events, it records one or more of them based on the following precedence, where 1 is the highest precedence:



1. Supervisor-trace event
2. Breakpoint- (from **mark** or **fmark** instruction, or from a breakpoint register), branch-, call-, or return-trace event
3. Instruction-trace event

When multiple trace events are detected, the processor may not signal each event; however, it will signal at least the one with the highest precedence.

## TRACE-HANDLING ACTION

Once a trace event has been signaled, the processor determines how to handle the trace event, according to the setting of the trace-enable and trace-fault-pending flags in the process controls and to other events that might occur simultaneously with the trace event such as an interrupt or a non-trace fault.

The following sections describe how the processor handles trace events for various situations.

### Normal Handling of Trace Events

Prior to executing an instruction, the processor performs the following action regarding trace events:

1. The processor checks the state of the trace-fault pending flag. If this flag is clear, the processor begins execution of the next instruction. If the flag is set, the processor performs the following actions.
2. The processor checks the state of the trace-enable flag. If the trace-enable flag is clear, the processor clears any trace event flags that have been set, prior to starting execution of the next instruction. If the trace-enable flag is set, the processor performs the following action.
3. The processor signals a trace fault and begins the fault handling action, as described in Chapter 12.

### Prereturn-Trace Handling

The processor handles a prereturn-trace event the same as described above except when it occurs at the same time as a non-trace fault. Here, the non-trace fault is handled first.

On returning from the fault handler for the non-trace fault, the processor checks the prereturn-trace flag in register r0. If this flag is set, the processor generates a prereturn-trace event, then handles it as described above.

### Tracing and Interrupt Handlers

When the processor invokes an interrupt handler to service an interrupt, it disables tracing. It does this by saving the current state of the process controls, then clearing the trace-enable and trace-fault-pending flags in the current process controls.



On returning from the interrupt handler, the processor restores the process controls to the state they were in prior to handling the interrupt, which restores the state of the trace-enable and trace-fault-pending flags. If these two flags were set prior to calling the interrupt handler, a trace fault will be signaled on the return from the interrupt handler.

### Tracing and Fault Handlers

The processor can invoke a fault handler with either an implicit local call or an implicit supervisor call. On a local call, the trace-enable and trace-fault-pending flags are neither saved on the call nor restored on the return. The state of these flags on the return is thus dependent on the action of the fault handler.

On a supervisor call, the trace-enable and trace-fault-pending flags are saved, as part of the saved process controls, and restored on the return. So, if these two flags were set prior to calling the fault handler, a trace fault will be signaled on the return from the fault handler.

#### NOTE

On a return from an interrupt handler, the trace-fault-pending flag is restored. If this flag is set as a result of the handler's **ret** instruction (i.e., indicating a return trace event), the detected trace event is lost.

The action described above is also true on a return from a fault handler, when the fault handler has been called with an implicit supervisor call.

---

---

## Instruction Reference

17

---

## CHAPTER 17

# INSTRUCTION REFERENCE

This chapter provides detailed information about each of the instructions for the 80960MC processor. To provide quick access to information on a particular instruction, the instructions are listed alphabetically by assembly-language mnemonic. An explanation of the format and abbreviations used in this chapter is given in the following section.

### INTRODUCTION

The information in this chapter is oriented toward programmers who are writing assembly-language code for the 80960MC processor. The information provided for each instruction includes the following:

- Alphabetic reference
- Assembly-language mnemonic and name
- Assembly-language format
- Description of the instruction's operation
- Action the instruction carries out when executed (generally presented in the form of an algorithm)
- Faults that can occur during execution
- Assembly-language example
- Opcode and instruction format
- Related instructions

Additional information about the instruction set can be found in the following chapters and appendices in this manual:

- Chapter 6 — Summary of the instruction set by group and description of the assembly-language instruction format
- Appendix A — Instruction Quick Reference
- Appendix B — Machine-Level Instruction Formats
- Appendix C — Instruction Timing

### NOTATION

To simplify the presentation of information about the instructions, a simple notation has been adopted in this chapter. The following paragraphs describe this notation.

## Alphabetic Reference

The instructions are listed alphabetically by assembly-language mnemonic. If several instructions are related and fall together alphabetically, they are described as a group on a single page.

The reference at the top of each page gives the assembly-language mnemonics for the instructions covered on that page (e.g., **subc**). Occasionally, there are so many instructions covered on the page that it is not practical to give all the mnemonics in the page reference. In these cases, the name of the instruction group is given in capital letters (e.g., **BRANCH** or **FAULT IF**).

A box around the alphabetic reference (such as **addr, addr1**) indicates that the instruction or group of instructions are extensions to the 80960 architecture instruction set.

## Mnemonic

The Mnemonic section gives the complete mnemonic (in bold-face type) and instruction name for each instruction covered on the page, for example:

**subi**                  Subtract Integer

## Format

The Format section gives the assembly-language format of the instruction and the type of operands allowed. The format is given in two or three lines. The following is an example of a two line format:

<b>sub*</b>	<i>src1,</i>	<i>src2,</i>	<i>dst</i>
	reg/lit	reg/lit	reg

The first line gives the assembly-language mnemonic (bold-face type) and the operands (italics). When the format is used for two or more instructions, an abbreviated form of the mnemonic is used. The "\*" sign at the end of the mnemonic indicates that the mnemonic has been abbreviated.

The operand names are designed to describe the functions of the operands (e.g., *src*, *len*, *mask*).

The second line of the format shows what is allowed to be entered for each operand. The notation used on this line is as follows:

reg	Global (g0 ... g15) or local (r0 ... r15) register
freg	Global (g0 ... g15) or local (r0 ... r15) register, or floating-point (fp0 ... fp3) register, where the registers contain floating-point numbers
lit	Integer or ordinal literal of the range 0 ... 31
flit	Floating-point literal of value 1.0 or 0.0
disp	Signed displacement of range $-2^{22} \dots (2^{22} - 1)$

mem Address defined with the full range of addressing modes

In some cases, a third line will be added to show specifically what will be in a register or memory location. For example, it may be useful to know that a register is to contain an address. The notation used in this line is as follows:

addr Address  
efa Effective address  
SS Segment selector

## Description

The Description section describes what the instruction does and the functions of the operands. It also gives programming hints when appropriate.

## Action

The Action section gives an algorithm written in a pseudo-code that describes in detail what actions the processor takes when executing the instruction and the precise order of these actions. The main purpose of this section is to show the possible side effects of the instruction. The following is an example of the action algorithm for the **alterbit** instruction:

```
if (AC.cc and 2#010#) = 0
  then dst ← src and not (2^(bitpos mod 32));
  else dst ← src or 2^(bitpos mod 32);
end if;
```

In these action statements, the term AC.cc means the condition-code bits in the arithmetic controls. The notation 2#value# means that the value enclosed in the "##" signs is in base 2.

## Faults

The Faults section lists the faults that can be signaled as the result of execution of the instruction. Faults listed with all-capital letters refer to a group of faults; faults listed with initial-capital letters refer to a specific fault.

All instructions can signal a group of general faults which are referred to as STANDARD FAULTS. The list of standard faults is as follows:

### STANDARD FAULTS

Trace Instruction  
Trace Process  
Process Time Slice  
Machine Bad Access  
Virtual Memory Segment  
Virtual Memory PTD  
Virtual Memory PTE



## Protection Length

## Protection Page Rights

Note that the virtual memory and protection faults listed above can occur on instructions that only access registers. Here, they can occur as a result of the memory access to fetch the instruction.

The following list shows the various fault groups and the individual faults in each group:

## TRACE FAULTS

Instruction Trace  
Branch Trace  
Call Trace  
Return Trace  
Prereturn Trace  
Supervisor Trace  
Breakpoint Trace

## OPERATION

Invalid Opcode  
Invalid Operand

## ARITHMETIC

Integer Overflow  
Arithmetic Zero-Divide

## FLOATING-POINT

Floating Overflow  
Floating Underflow  
Floating Invalid-Operation  
Floating Zero-Divide  
Floating Inexact  
Floating Reserved-Encoding

## CONSTRAINT

Constraint Range  
Invalid SS

## VIRTUAL MEMORY

Invalid Segment  
Invalid Page-Table-Directory-Entry (PTDE)  
Invalid Page-Table-Entry (PTE)

## PROTECTION

Segment Length  
Page Rights

## MACHINE

Bad Access

## STRUCTURAL

Control  
Dispatch  
IAC

addc

## TYPE

Type Mismatch  
Contents

Format:

## PROCESSOR

Time Slice

Description:

## DESCRIPTOR

Invalid Descriptor

## EVENT

Event Notice

## Example

The Example section gives an assembly-language example of an application of the instruction.

## Opcode and Instruction Format

The Opcode and Instruction Format section gives the opcode and machine language instruction format for each instruction, for example:

subi 593 REG

The opcode is given in hexadecimal format.

The machine language format is one of four possible formats: REG, COBR, CTRL, and MEM. Refer to Appendix B for more information on the machine-language instruction formats.

## See Also

The See Also section gives the mnemonics of related instructions, which can then be looked up alphabetically in this chapter for comparison. For instructions that are grouped on one page (such as **addr** and **addr1**), only the first mnemonic is given.

## INSTRUCTIONS

This section contains reference information on the processor's instructions. It is arranged alphabetically by instruction or instruction group.

**addc**

**Mnemonic:**     **addc**     Add Ordinal With Carry

<b>Format:</b>	<b>addc</b>	<i>src1,</i>	<i>src2,</i>	<i>dst</i>
		reg/lit	reg/lit	reg

**Description:** Adds the *src2* and *src1* values, and bit 1 of the condition code (used here as a carry in), and stores the result in *dst*. If the ordinal addition results in a carry, bit 1 of the condition code is set; otherwise, bit 1 is cleared. If integer addition results in an overflow, bit 0 of the condition code is set; otherwise, bit 0 is cleared. Regardless of the results of the addition, bits 0 and 1 of the arithmetic controls are always written.

The **addc** instruction can be used for either ordinal or integer arithmetic. The instruction does not distinguish between ordinal and integer source operands. Instead, the processor evaluates the result for both data types and sets bits 0 and 1 of the condition code accordingly.

An integer overflow fault is never signaled with this instruction.

**Action:** # Let the value of the condition code be xCx.  
 $dst \leftarrow src2 + src1 + C;$   
 $AC.cc \leftarrow 2\#0CV\#;$   
 # C is carry from ordinal addition.  
 # V is 1 if integer addition would have generated an overflow;

Faults: STANDARD

```

Example:      # Example of double-precision arithmetic
                  # Assume 64-bit source operands
                  # in g0,g1 and g2,g3
cmpo 1, 0        # clears Bit 1 (carry bit) of
                  # the AC.cc
addc g0, g2, g0   # add low-order 32 bits;
                  #  $g0 \leftarrow g2 + g0 + \text{Carry Bit}$ 
addc g1, g3, g1   # add high-order 32 bits;
                  #  $g1 \leftarrow g3 + g1 + \text{Carry Bit}$ 
                  # 64-bit result is in g0, g1

```

**Opcode:**      **addc**      5B0      REG

**See Also:**      **addo, subc**

addi, addo

Mnemonic:    **addi**            Add Integer  
                 **addo**            Add Ordinal

Format:        **add\***        *src1*,        *src2*,        *dst*  
                                 reg/lit        reg/lit        reg

Description:    Adds the *src2* and *src1* values and stores the result in *dst*.

Action:         *dst* ← *src2* + *src1*;

Faults:         STANDARD

Integer Overflow

Refer to discussion of faults at the beginning of this chapter.  
Result is too large for destination format. This fault is signaled only when executing the **addi** instruction and if both of the following conditions are met: (1) the integer-overflow mask in the arithmetic-controls registers is clear and (2) the source operands have like signs and the sign of the result operand is different than the signs of the source operands.

Example:        **addi** r4, g5, r9        # r9 ← g5 + r4

Opcode:        **addi**        591        REG  
                 **addo**        590        REG

See Also:       **addc, addr, subi, subo**

\* Indicates floating invalid-operation exception  
F Means finite-real number

When the sum of two operands with opposite signs is zero, the result is +0, except for the round toward -∞ mode, in which case, the result is -0. When zero is added to itself (e.g. *src1* + *src1*, where *src1* is 0), the result retains the sign of the source.

Action:         *dst* ← *src2* + *src1*;

**addr, addrl**

**Mnemonics:**    **addr**            Add Real  
                   **addrl**          Add Long Real

**Format:**        **addr\***        *src1*,        *src2*,        *dst*  
    freg/flit        freg/flit        freg

**Description:**    Adds the *src2* and *src1* values and stores the result in *dst*.

For the **addrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						
Src2		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	NaN
	$-F$	$-\infty$	$-F$	src2	src2	$\pm F$ or $\pm 0$	$+\infty$	NaN
	$-0$	$-\infty$	src1	$-0$	$\pm 0$	src1	$+\infty$	NaN
	$+0$	$-\infty$	src1	$\pm 0$	$+0$	src1	$+\infty$	NaN
	$+F$	$-\infty$	$\pm F$ or $\pm 0$	src2	src2	$+F$	$+\infty$	NaN
	$+\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**Notes:**

- F    Means finite-real number
- \*    Indicates floating invalid-operation exception

When the sum of two operands with opposite signs is zero, the result is +0, except for the round toward -∞ mode, in which case, the result is -0. When zero is added to itself (e.g. *src1* + *src1*, where *src1* is 0), the result retains the sign of the source.

**Action:**         $dst \leftarrow src2 + src1;$

**addr, addr1**

<b>Faults:</b>	STANDARD	Refer to the discussion of faults at the beginning of this chapter.
	Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.
	The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.	
	Floating Overflow	Result is too large for destination format.
	Floating Underflow	Normalized result is too small for destination format.
	Floating Invalid Operation	Source operands are infinities of unlike sign.
		One or more operands is an SNaN value.
	Floating Inexact	Result cannot be represented exactly in destination format.
		Floating overflow occurred and the overflow exception was masked.
<b>Example:</b>	addr1 g6, g8, fp3    #fp3 ← g6, g7 + g8, g9	
<b>Opcode:</b>	addr        78F        REG	
	addr1      79F        REG	
<b>See Also:</b>	addi, subr	



alterbit

Mnemonic: alterbit Alter Bit

Format: alterbit bitpos, src, dst  
reg/lit reg/lit reg

Description: Copies the *src* value to *dst* with one bit altered. The *bitpos* operand specifies the bit to be changed; the condition code determines the value the bit is to be changed to. If the condition code is *X1X<sub>2</sub>*, the selected bit is set; otherwise, it is cleared.

Action: if (AC.cc and 2#010#) = 0  
then *dst* ← *src* and not (2<sup>(*bitpos* mod 32)</sup>);  
else *dst* ← *src* or 2<sup>(*bitpos* mod 32)</sup>;  
end if;

Faults: STANDARD

Example: # assume AC.cc = 010  
alterbit 24, g4, g9  
# g9 ← g4, with bit 24 set

Opcode: alterbit 58F REG

See Also: checkbit, clearbit, notbit, setbit

and, andnot

Mnemonics:	and andnot	And And Not		
Format:	and	src1, reg/lit	src2, reg/lit	dst reg
	andnot	src1, reg/lit	src2, reg/lit	dst reg
Description:	Performs a bitwise AND ( <b>and</b> instruction) or AND NOT ( <b>andnot</b> instruction) operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> . Note in the action expressions below, the <i>src2</i> operand comes first, so that with the <b>andnot</b> instruction the expression is evaluated as $\{src2 \text{ andnot } (src1)\}$ rather than $\{src1 \text{ andnot } (src2)\}$ .			
Action:	and:	$dst \leftarrow src2 \text{ and } src1;$		
	andnot:	$dst \leftarrow src2 \text{ and not } (src1);$		
Faults:	STANDARD			
Example:	and 0x17, g8, g2 # g2 $\leftarrow$ g8 AND 0x17 andnot r3, r12, r9 # r9 $\leftarrow$ r12 AND NOT r3			
Opcode:	and	581	REG	
	andnot	582	REG	
See Also:	nand, nor, not, notand, notor, or, ornot, xnor, xor			

atadd

Mnemonic:    **atadd**           Atomic Add

Format:       **atadd**       *src/dst*,    *src*,       *dst*  
                              reg        reg/lit     reg  
                              addr       reg/lit     reg

Description:   Adds the *src* value (full word) to the value in the memory location specified with the *src/dst* operand. The initial value from memory is stored in *dst*.

The read and write of memory are done atomically (i.e., other processors are prevented from accessing the word of memory specified with the *src/dst* operand until the operation has been completed).

The memory location in *src/dst* is the address of the first byte (least significant byte) of the word. The address is automatically aligned to a word boundary. (Note that the *src/dst* operand maps to the *src1* operand of the REG machine-code format. Refer to Appendix B for a description of the REG format.)

Action:       tempa ← *src/dst* and not (3); # force alignment to word boundary  
              temp ← atomic\_read (tempa);  
              atomic\_write (tempa) ← temp + *src*;  
              *dst* ← temp;

Faults:       STANDARD

Example:      **atadd** r8, r2, r11   # r8 ← r2 + address r8,  
                                  # where r8 specifies the  
                                  # address of a word in  
                                  # memory; r11 ← initial  
                                  # value stored at address  
                                  # r8 in memory

Opcode:       **atadd**       612       REG

See Also:      **atmod**

atanr, atanrl

Mnemonics:    **atanr**            Arc tangent Real  
                  **atanrl**          Arc tangent Long Real

Format:        **atanr\***    *src1*,        *src2*,        *dst*  
                                  freg/flit        freg/flit        freg

Description:    Calculates the arc tangent of the quotient of *src2/src1* and stores the result in *dst*. The result is returned in radians and is in the range of  $-\pi$  to  $+\pi$ , inclusive. The sign of the result is always the sign of *src2*.

For the **atanrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

These instructions are commonly used as part of an algorithm to convert rectangular coordinates to polar coordinates. They can also be used to implement the FORTRAN intrinsic functions ATAN and ATAN2. If *src1* is the floating-point literal value +1.0, then these instructions return a result in the range of  $-\pi/2$  to  $+\pi/2$ .

The following table gives the range of results for various values of *src2* and *src1*, assuming that neither overflow nor underflow occurs.

		Src1						
		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	$-3\pi/4$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4$	NaN
	$-F$	$-\pi$	$-\pi$ to $-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$ to $-0$	$-0$	NaN
	$-0$	$-\pi$	$-\pi$	$-\pi$	$-0$	$-0$	$-0$	NaN
Src2	$+0$	$+\pi$	$+\pi$	$+\pi$	$+0$	$+0$	$+0$	NaN
	$+F$	$+\pi$	$+\pi$ to $+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$ to $+0$	$+0$	NaN
	$+\infty$	$+3\pi/4$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/4$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F    Means finite-real number.

atanr, atanrl

Action:  $dst \leftarrow \arctan (src2/src1);$

Faults: STANDARD Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Underflow Result is too small for destination format.

Floating Invalid Operation One or more operands are an SNaN value.

Floating Inexact Result cannot be represented exactly in destination format.

Example: atanrl g8, g10, fp3 # fp3 ← # arctan (g10,g11/g8,g9)  
atanrl 1.0, g0, g0 # g0,g1 ← arctan (g0,g1)

Opcode:	atanr	680	REG			
	atanrl	690	REG			
See Also:	tanr					

atmod

Mnemonic:	atmod	Atomic Modify				
Format:	atmod	src, reg addr	mask, reg/lit	src/dst reg		
Description:		<p>Copies the <i>src/dst</i> value into the memory location specified in <i>src</i>. The bits set in the <i>mask</i> operand select the bits to be modified in memory. The initial value from memory is stored in <i>src/dst</i>.</p> <p>The read and write of memory are done atomically (i.e., other processors are prevented from accessing the word of memory specified with the <i>src/dst</i> operand until the operation has been completed).</p> <p>The memory location in <i>src</i> is the address of the first byte (least significant byte) of the word to be modified. The address is automatically aligned to a word boundary.</p>				
Action:		<p>tempa ← <i>src</i> and not (3); # force alignment to word boundary temp ← atomic_read (tempa); atomic_write (tempa) ← (<i>src/dst</i> and mask) or (temp and not(mask)); <i>src/dst</i> ← temp;</p>				
Faults:		STANDARD				
Example:		<pre>atmod g5, g7, g10 # g5 ← g5 masked by g7,                   # where g5 specifies the                   # address of a word in                   # memory;                   # g10 ← initial value                   # stored at address g5                   # in memory</pre>				
Opcode:	atmod	610	REG			
See Also:	atadd					



**b, bx**

<b>Mnemonic:</b>	<b>b</b>	Branch
	<b>bx</b>	Branch Extended
<b>Format:</b>	<b>b</b>	<i>targ</i> disp

	<b>bx</b>	<i>targ</i> mem
--	-----------	--------------------

**Description:** Branches to the instruction specified with the *targ* operand. When using the Intel 80960MC Assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

With the **b** instruction, the IP specified with the *targ* operand can be no farther than  $-2^{23}$  to  $(2^{23} - 4)$  bytes from the current IP.

The **bx** instruction performs the same operation as the **b** instruction except that the target instruction can be farther than  $-2^{23}$  to  $(2^{23} - 4)$  bytes from the current IP. Here, the *targ* operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect branching can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.

**NOTE**

At the machine level, the **b** instruction uses the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statement for the **b** instruction), which can range from  $-2^{21}$  to  $(2^{21} - 1)$ . To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

**b, bx**

To allow labels to be used in the assembly-language version of the **b** instruction, the Intel 80960MC Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine-instruction format:

$$\text{displacement} = (\text{targ} - \text{IP})/4$$

For further information about the CTRL instruction format, refer to Appendix B.

**Action:** **b:**  $\text{IP} \leftarrow \text{IP} + \text{displacement};$  # resume execution at new IP

**bx:**  $\text{IP} \leftarrow \text{targ};$  # resume execution at new IP

**Faults:** STANDARD

**Example:** `b xyz # IP ← xyz;`

`bx 1332 (ip) # IP ← IP + 1332;`  
 # this example uses ip-relative  
 # addressing.

**Opcode:** **b** 08 CTRL  
**bx** 84 MEM

**See Also:** **bal, balx, BRANCH IF, COMPARE INTEGER AND BRANCH, COMPARE ORDINAL AND BRANCH**

**NOTE**

At the machine level, the **bal** instruction uses the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by displacement in the following section statement for the **bal** instruction), which can range from -2<sup>31</sup> to 2<sup>31</sup> - 1. To determine the IP of the target instruction, the processor converts this displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

## bal, balx

xd, d

**Mnemonic:** **bal** Branch And Link  
**balx** Branch And Link Extended

**Format:** **bal** *targ*  
*disp*

**balx** *targ, dst*  
*mem reg*

**Description:** Stores the address of the next instruction (the instruction following the **bal** or **balx** instruction) and branches to the instruction specified with the *targ* operand. When using the Intel 80960MC Assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

With the **bal** instruction, the address of the next instruction is stored in register g14. The *targ* operand value can be no farther than  $-2^{23}$  to  $(2^{23} - 4)$  bytes from the current IP.

The **balx** instruction performs almost the same operation as the **bal** instruction except that the address of the next instruction is stored in *dst*. With the **balx** instruction, the target instruction can be farther than  $-2^{23}$  to  $(2^{23} - 4)$  bytes from the current IP. Here, the *targ* operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect branching can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.

### NOTE

At the machine level, the **bal** instruction uses the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statement for the **bal** instruction), which can range from  $-2^{21}$  to  $(2^{21} - 1)$ . To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.



## bbc, bbs

**Mnemonic:** **bbc** Check Bit and Branch If Clear  
**bbs** Check Bit and Branch If Set

**Format:** **bb\*** *bitpos*, *src*, *targ*  
reg/lit reg disp

**Description:** Checks the bit in *src* (designated by *bitpos*) and sets the condition code in the arithmetic controls according to the value found. The processor then performs a conditional branch to the instruction specified with the *targ* operand, according on the state of the condition code. When using the Intel 80960MC Assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

For the **bbc** instruction, if the selected bit in **src** is clear, the processor sets the condition code to 010<sub>2</sub> and branches to the instruction specified with the *targ* operand; otherwise, it sets the condition code to 000<sub>2</sub> and goes to the next instruction.

For the **bbs** instruction, if the selected bit is set, the processor sets the condition code to 010<sub>2</sub> and branches to *targ*; otherwise, it sets the condition code to 000<sub>2</sub> and goes to the next instruction.

The *targ* operand can be no farther than -2<sup>12</sup> to (2<sup>12</sup> - 4) bytes from the current IP.

### NOTE

At the machine level, the **bbc** and **bbs** instructions use the COBR instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statement), which can range from -2<sup>10</sup> to (2<sup>10</sup> - 1). To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels to be used in the assembly-language versions of the **bbc** and **bbs** instructions, the Intel 80960MC Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

$$displacement = (targ - IP)/4$$

For further information about the COBR instruction format, refer to Appendix B.

# bbc, bbs

**Action:**      **bbc:**

```

if (src and 2^(bitpos mod 32)) = 0
then AC.cc ← 2#010#;
    IP ← IP + 4 + (displacement * 4);
    # resume execution at the new IP
else AC.cc ← 2#000#;
    IP ← IP + 4; # resume execution at the next IP
end if;

```

**bbs:**

```

if (src and 2^(bitpos mod 32)) = 1
then AC.cc ← 2#010#;
    IP ← IP + 4 + (displacement * 4);
    # resume execution at the new IP
else AC.cc ← 2#000#;
    IP ← IP + 4; # resume execution at the next IP
end if;

```

**Faults:**      STANDARD

**Example:**    # assume bit 10 of r6 is clear  
                   bbc 10, r6, xyz    # bit 10 of r6 is checked  
                                       # and found clear;  
                                       # AC.cc ← 010  
                                       # IP ← xyz;

**Opcode:**      **bbc**      30      COBR  
                   **bbs**      37      COBR

**See Also:**      chkbit



## BRANCH IF

Mnemonics:	<b>be</b>	Branch If Equal
	<b>bne</b>	Branch If Not Equal
	<b>bl</b>	Branch If Less
	<b>ble</b>	Branch If Less Or Equal
	<b>bg</b>	Branch If Greater
	<b>bge</b>	Branch If Greater Or Equal
	<b>bo</b>	Branch If Ordered
	<b>bno</b>	Branch If Unordered

**Format:**      **b\***      *targ*  
                                 *disp*

**Description:** Branches to the instruction specified with the *targ* operand, according to the state of the condition code in the arithmetic controls. When using the Intel 80960MC Assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

For all branch-if instructions, except the **bno** instruction, the processor branches to the instruction specified with the *targ* operand, if the logical AND of the condition code and the mask-part of the opcode is not zero. Otherwise, it goes to the next instruction.

For the **bno** instruction, the processor branches to the instruction specified with *targ*, if the logical AND of the condition code and the mask-part of the opcode is zero. Otherwise, it goes to the next instruction.

The *targ* operand value can be no farther than  $-2^{23}$  to  $(2^{23} - 4)$  bytes from the current IP.

### NOTE

At the machine level, the branch-if instructions use the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statements), which can range from  $-2^{21}$  to  $(2^{21} - 1)$ . To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

## BRANCH IF

To allow labels to be used in the assembly-language version of the branch-if instructions, the Intel 80960MC Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

$$\text{displacement} = (\text{targ} - \text{IP})/4$$

For further information about the CTRL instruction format, refer to Appendix B.

The following table shows the condition-code mask for each instruction:

Instruction	Mask	Condition
<b>bno</b>	000	Unordered
<b>bg</b>	001	Greater
<b>be</b>	010	Equal
<b>bge</b>	011	Greater or equal
<b>bl</b>	100	Less
<b>bne</b>	101	Not equal
<b>ble</b>	110	Less or equal
<b>bo</b>	111	Ordered

For the **bno** instruction (unordered), the branch is taken if the condition code is equal to  $000_2$ .

The mask is in bits 0-2 of the opcode.

**Action:**

For All Instructions Except **bno**:

```
if (mask and AC.cc) ≠ 2#000#  
  then IP ← IP + displacement; # resume execution at new IP  
end if;
```

**bno**:

```
if AC.cc = 2#000#  
  then IP ← IP + displacement; # resume execution at new IP  
end if;
```

BRANCH IF

Faults: STANDARD

Example: # assume (AC.cc AND 100) ≠ 0  
bl xyz # IP ← xyz;

Opcode:

be	12	CTRL
bne	15	CTRL
bl	14	CTRL
ble	16	CTRL
bg	11	CTRL
bge	13	CTRL
bo	17	CTRL
bno	10	CTRL

See Also: b, bx

Instruction	Condition Code
bno	Unordered
bg	Greater
bne	Not Equal
bge	Greater or Equal
bl	Less
ble	Less or Equal
bo	Ordered

For the bno instruction (unordered), the branch is taken if the condition code is equal to 000.

The mask is in bits 0-2 of the opcode.

For All Instructions Except bno:

if (mask and AC.cc) ≠ 2#000#  
then IP ← IP + displacement; # resume execution at new IP  
end if;

bno:

if AC.cc = 2#000#  
then IP ← IP + displacement; # resume execution at new IP  
end if;

Action:

**call**

**Mnemonic:**    **call**            **Call**

**Format:**        **call**            *targ*  
   *disp*

**Description:**    Calls a new procedure. The *targ* operand specifies the IP of the first instruction of the called procedure. When using the Intel 80960MC Assembler, the *targ* operand must be a label.

In executing this instruction, the processor performs a local call operation as described in Chapter 4 in the section titled "Local Calls." As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. The processor then goes to the instruction specified with the *targ* argument and begins execution of the new procedure.

The *targ* operand can be no farther than  $-2^{23}$  to  $(2^{23} - 4)$  bytes from the current IP.

**NOTE**

At the machine level, the **call** instruction uses the CTRL instruction format. With this format, the first instruction of the called procedure is specified by means of a word-displacement (represented by *displacement* in the following action statement), which can range from  $-2^{21}$  to  $(2^{21} - 1)$ . To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels to be used in the assembly-language version of the **call** instruction, the Intel 80960MC Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

$$displacement = (targ - IP)/4$$

For further information about the CTRL instruction format, refer to Appendix B.

## call

**Action:** wait for any uncompleted instructions to finish;  
 $\text{temp} \leftarrow (\text{SP} + 63) \text{ and not } (63); \# \text{ round to next boundary}$   
 $\text{RIP} \leftarrow \text{IP};$   
 if register\_set\_available  
   **then** allocate as new frame;  
   **else** save a register\_set in memory at its FP;  
     allocate as new frame;  
   # local register references now refer to new frame  
 $\text{IP} \leftarrow \text{IP} + \text{displacement};$   
 $\text{PFP} \leftarrow \text{FP};$   
 $\text{FP} \leftarrow \text{temp};$   
 $\text{SP} \leftarrow \text{temp} + 64;$

**Faults:** STANDARD

**Example:** `call xyz # IP ← xyz`

**Opcode:** `call 09 CTRL`

**See Also:** `bal, calls, callx`

## calls

**Mnemonic:**    **calls**            Call System

**Format:**        **calls**            *targ*  
   *reg/lit*

**Description:**    Calls a system procedure. The *targ* operand gives the number of the procedure being called.

For this instruction, the processor performs the system call operation described in Chapter 4 in the section titled "System Calls." The *targ* operand provides an index to an entry in the system procedure table. From this entry, the processor gets the IP of the called procedure.

The procedure called can be either a local procedure or a supervisor procedure, depending on the entry type in the procedure table. If it is a supervisor procedure, the processor also switches to supervisor mode (if it is not already in this mode).

As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. If the processor switches to the supervisor mode, the new stack frame is created on the supervisor stack.

**Action:**            **if** *targ* > 259 **then** raise Protection Length Fault;  
                                 wait for any uncompleted instructions to finish;  
                                 *temp\_p\_e* ← memory (SPTSS, 48 + (4 \* *targ*));  
                                 # SPTSS is SS to system procedure table from PRCB  
                                 *RIP* ← *IP*;  
                                 *IP* ← *temp\_p\_e.address*; **if** (*temp\_p\_e.type* = local) **or**  
                                 *execution\_mode* = supervisor  
                                 **then** *temp* ← (*SP* + 63) **and not**(63);  
                                 *tempRRR* ← 2#000#;  
                                 **else** *temp* ← memory (SPTSS, 12); # supervisor call  
                                 *tempRRR* ← 2#01T#; # T is process\_controls.T  
                                 *execution\_mode* ← supervisor;  
                                 process\_controls.T ← *temp.T*;  
                                 **endif**;



calls

		Mnemonic:	calls	Call System
		Format:	calls	target register
		Description:	Calls the procedure specified by the target register. The procedure is located in the system procedure table. The register provides an index to an entry in the system procedure table. From this entry, the processor gets the IP of the called procedure.	
		if frame_available then allocate as new frame; else save a frame in memory at its FP; allocate as new frame; # local register references now refer to new frame endif; PFP ← FP; LO.RRR ← tempRRR; FP ← temp; SP ← temp + 64;		
Faults:	STANDARD			
Example:	calls r12   # IP ← value obtained from # procedure table for procedure # number given in r12			
Opcode:	calls	660	REG	
See Also:	bal, call, callx			

## callx

**Mnemonic:** **callx** Call Extended

**Format:** **callx** *targ*  
*mem*

**Description:** Calls a new procedure. The *targ* operand specifies the IP of the first instruction of the called procedure. When using the Intel 80960MC Assembler, the *targ* operand must be a label.

In executing this instruction, the processor performs a local call operation as described in Chapter 4 in the section titled "Local Calls." As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. The processor then goes to the instruction specified with the *targ* argument and begins execution of the new procedure.

This instruction performs the same operation as the **call** instruction except that the target instruction can be farther than  $-2^{23}$  to  $(2^{23} - 4)$  bytes from the current IP.

The *targ* operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect calls can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.

**Action:**

```

wait for any uncompleted instructions to finish;
temp ← (SP + 63) and not (63); # round to next boundary
RIP ← IP;
if register_set_available
    then allocate as new frame;
    else save a register_set in memory at its FP;
        allocate as new frame;
# local register references now refer to new frame
endif;
IP ← targ;
PFP ← FP;
FP ← temp;
SP ← temp + 64;

```

callx

Faults: STANDARD

Example: callx (g5) # IP ← (g5), where the address  
# in g5 is the address of the new  
# procedure

Opcode: callx 86 MEM

See Also: call, calls

This instruction performs the same operation as the call instruction except that the target instruction can be farther than -2<sup>32</sup> to (-2<sup>32</sup> - 4) bytes from the current IP.

The *rax* operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect calls can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.

Action:

```
wait for any uncompleted instructions to finish;
temp ← (SP + 03) and not (03); # round to next boundary
RIP ← IP;
if register_set_available
then allocate as new frame;
else save a register_set in memory at its FP;
allocate as new frame;
# local register references now refer to new frame;
endif;
IP ← temp;
RBP ← FP;
FP ← temp;
SP ← temp + 04;
```

# chkbit

**Mnemonic:** **chkbit** Check Bit

**Format:** **chkbit** *bitpos*, *src*  
reg/lit reg/lit

**Description:** Checks the bit in *src* designated by *bitpos* and sets the condition code according to the value found. If the bit is set, the condition code is set to 010<sub>2</sub>; if the bit is clear, the condition code is set to 000<sub>2</sub>.

**Action:** if (*src* and 2<sup>(*bitpos* mod 32)</sup>) = 0  
then AC.cc ← 2#000#;  
else AC.cc ← 2#010#;  
end if;

**Faults:** STANDARD

**Example:** `chkbit 13, g8 # checks bit 13 in g8`

**Opcode:** **chkbit** 5AE REG

**See Also:** `alterbit`, `clrbit`, `notbit`, `setbit`

Class	Classification
000	Reserved operand
001	Reserved operand
010	Reserved operand
011	Reserved operand
100	Reserved operand
101	Reserved operand
110	Reserved operand
111	Reserved operand

The "s" bit is set to the sign of the *src* operand.

Refer to Chapter 7 for a discussion of the different real number classifications.

**classr, classrl**

**Mnemonic:**    **classr**        Classify Real  
                 **classrl**      Classify Long Real

**Format:**        **classr\***    *src*  
                                  freg/flit

**Description:**    Checks the classification of the real number in *src* and stores the class in arithmetic-status bits (3 through 6) of the arithmetic controls.

For the **classrl** instruction, if the *src* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the setting of the arithmetic-status bits depending on the classification of the operand.

AStatus	Classification
s000	Zero
s001	Denormalized number
s010	Normal finite number
s011	Infinity
s100	Quiet NaN
s101	Signaling NaN
s110	Reserved operand

The "s" bit is set to the sign of the *src* operand.

Refer to Chapter 7 for a discussion of the different real number classifications.

**classr, classrl****Action:**

```
s ← sign_of(src)
if src = 0
    then arithmetic_status ← s000;
elseif src = denormalized
    then arithmetic_status ← s001;
elseif src = normal finite
    then arithmetic_status ← s010;
elseif src = ∞
    then arithmetic_status ← s011;
elseif src = QNaN
    then arithmetic_status ← s100;
elseif src = SNaN
    then arithmetic_status ← s101;
elseif src = reserved operand
    then arithmetic_status ← s110;
end if
```

**Faults:**

STANDARD

Refer to the discussion of faults at the beginning of this chapter.

None of the floating-point exceptions can be raised.

**Example:**

```
classrl g12    # classifies long real in g12,g13
```

**Opcode:**

<b>classr</b>	68F	REG
<b>classrl</b>	69F	REG



clrbt

Mnemonic:	clrbt	Clear Bit
Format:	clrbt	bitpos, src, dst reg/lit reg/lit reg
Description:	Copies the <i>src</i> value to <i>dst</i> with one bit cleared. The <i>bitpos</i> operand specifies the bit to be cleared.	
Action:	$dst \leftarrow src \text{ and not}(2^{(bitpos \bmod 32)})$ ;	
Faults:	STANDARD	
Example:	clrbt 23, g3, g6 # g6 ← g3 with bit 23 # cleared	
Opcode:	clrbt	58C REG
See Also:	alterbit, chkbit, notbit, setbit	

## cmpi, cmpo

**Mnemonics:** **cmpi** Compare Integer  
**cmpo** Compare Ordinal

**Format:** **cmp\*** *src1*, *src2*  
reg/lit reg/lit

**Description:** Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison. The following table shows the setting of the condition code for the three possible results of the comparison.

Condition Code	Comparison
100	$src1 < src2$
010	$src1 = src2$
001	$src1 > src2$

The **cmpi** instruction followed by one of the branch-if instructions is equivalent to one of the compare-integer-and-branch instructions. The latter method of comparing and branching produces more compact code; however, the former method can result in faster running code because it takes advantage of the processor's pipelined architecture. The same is true for the **cmpo** instruction and the compare-ordinal-and-branch instructions.

**Action:** if  $src1 < src2$  then AC.cc  $\leftarrow$  2#100#;  
elseif  $src1 = src2$  then AC.cc  $\leftarrow$  2#010#;  
else AC.cc  $\leftarrow$  2#001#;  
end if;

**Faults:** STANDARD

**Example:** cmpo 0x10, r9 # compare values in r9 and 0x10  
# and set AC.cc

**Opcode:** **cmpi** 5A1 REG  
**cmpo** 5A0 REG

**See Also:** cmpibe, cmpr, cmpdeci, cmpdeco

## cmpdeci, cmpdeco

**Mnemonics:** **cmpdeci** Compare and Decrement Integer  
**cmpdeco** Compare and Decrement Ordinal

**Format:** **cmpdec\*** *src1*, *src2*, *dst*  
 reg/lit reg/lit reg

**Description:** Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison. The *src2* operand is then decremented by one and the result is stored in *dst*.

The following table shows the setting of the condition code for the three possible results of the comparison.

Condition Code	Comparison
100	$src1 < src2$
010	$src1 = src2$
001	$src1 > src2$

These instructions are intended for use in ending iterative loops. For the **cmpdeci** instruction, integer overflow is ignored to allow looping down through the minimum integer values.

**Action:** if  $src1 < src2$  then AC.cc  $\leftarrow$  2#100#;  
 elseif  $src1 = src2$  then AC.cc  $\leftarrow$  2#010#;  
 elseif  $src1 > src2$  then AC.cc  $\leftarrow$  2#001#;  
 end if;  
 $dst \leftarrow src2 - 1$ ; #overflow suppressed for **cmpdeci**  
 # instruction

**Faults:** STANDARD

**Example:** `cmpdeci 12, g7, g1` # g7 and 12 are compared;  
 #  $g1 \leftarrow g7 - 1$

**Opcode:** **cmpdeci** 5A7 REG  
**cmpdeco** 5A6 REG

**See Also:** *cmpinco, cmpo*

**cmpinci, cmpinco**

**Mnemonics:** **cmpinci** Compare and Increment Integer  
**cmpinco** Compare and Increment Ordinal

**Format:** **cmpinc\*** *src1*, *src2*, *dst*  
 reg/lit reg/lit reg

**Description:** Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison. The *src2* operand is then incremented by one and the result is stored in *dst*.

The following table shows the setting of the condition code for the three possible results of the comparison.

Condition Code	Comparison
100	<i>src1</i> < <i>src2</i>
010	<i>src1</i> = <i>src2</i>
001	<i>src1</i> > <i>src2</i>

These instructions are intended for use in ending iterative loops. For the **cmpinci** instruction, integer overflow is ignored to allow looping up through the maximum integer values.

**Action:** **if** *src1* < *src2* **then** AC.cc ← 2#100#;  
**elseif** *src1* = *src2* **then** AC.cc ← 2#010#;  
**elseif** *src1* > *src2* **then** AC.cc ← 2#001#;  
**end if**;  
*dst* ← *src2* + 1; # overflow suppressed for **cmpinci**  
 # instruction

**Faults:** STANDARD

**Example:** `cmpinco r8, g2, g9` # g2 and r8 are compared;  
 # g9 ← g2 + 1

**Opcode:** **cmpinci** 5A5 REG  
**cmpinco** 5A4 REG

**See Also:** **cmpdeco, cmpo**

**cmpor, cmporl**

**Mnemonics:** **cmpor** Compare Ordered Real  
**cmporl** Compare Ordered Long Real

**Format:** **cmpor\*** *src1*, *src2*  
 freg/flit freg/flit

**Description:** Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison.

For the **cmporl** instruction, if the *src1* or *src2* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the setting of the condition code for the four possible results of the comparison.

Condition Code	Comparison
100	<i>src1</i> < <i>src2</i>
010	<i>src1</i> = <i>src2</i>
001	<i>src1</i> > <i>src2</i>
000	if either <i>src1</i> or <i>src2</i> is a NaN

The algorithm for these instructions checks the classification of the operands. If either is in the NaN class, the condition code is set to 000<sub>2</sub> and a floating invalid-operation exception is raised. The **cmpor** and **cmporl** instructions operate the same as the **cmpr** and **cmprl** instructions, except that the latter instructions do not signal an exception if a NaN value is detected.

If a floating-reserved-encoding fault occurs, the condition code results are undefined.

**Action:** **if** *src1* < *src2* **then** AC.cc ← 2#100#;  
**elseif** *src1* = *src2* **then** AC.cc ← 2#010#;  
**elseif** *src1* > *src2* **then** AC.cc ← 2#001#;  
**else** AC.cc ← 2#000#; # indicates one number is a NaN  
 raise floating invalid operation fault  
**end if;**



cmpor, cmporl

- Faults:

STANDARD

Refer to the discussion of faults at the beginning of this chapter.
- Floating Reserved Encoding

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- Floating Invalid Operation

One or more operands are a NaN value.

Example: cmporl g6, g12 # compare value in g12, g13  
# with value in g6, g7

Opcode:	cmpor	684	REG	Code
	cmporl	694	REG	
See Also:	cmpr, cmpi, BRANCH IF			000
				001
				010

The algorithm for these instructions checks the classification of the operands. If either is in the NaN class, the condition code is set to 000, but no fault is raised. The cmpr and cmpi instructions operate the same as the cmpor and cmporl instructions, except that the latter instructions raise an invalid-operand exception if a NaN value is detected.

If a floating-reserved-encoding fault occurs, the condition code results are undefined.

end if;  
else AC.cc ← 2#000#; # indicates one number is a NaN  
elseif src1 > src2 then AC.cc ← 2#001#;  
elseif src1 = src2 then AC.cc ← 2#010#;  
if src1 < src2 then AC.cc ← 2#100#;  
Action:



**cmpr, cmpri**

**Mnemonics:** **cmpr** Compare Real  
**cmpri** Compare Long Real

**Format:** **cmpr\*** *src1*, *src2*  
freg/flit freg/flit

**Description:** Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison. For the **cmpri** instruction, if the *src1* or *src2* operand references a global or local register, this register is the first (lowest numbered) of two successive registers.

The following table shows the setting of the condition code for the four possible results of the comparison.

Condition Code	Comparison
100	<i>src1</i> < <i>src2</i>
010	<i>src1</i> = <i>src2</i>
001	<i>src1</i> > <i>src2</i>
000	if either <i>src1</i> or <i>src2</i> is a NaN

The algorithm for these instructions checks the classification of the operands. If either is in the NaN class, the condition code is set to 000<sub>2</sub>, but no fault is raised. The **cmpr** and **cmpri** instructions operate the same as the **cmpor** and **cmpori** instructions, except that the latter instructions raise an invalid-operand exception if a NaN value is detected.

If a floating-reserved-encoding fault occurs, the condition code results are undefined.

**Action:**

```
if src1 < src2 then AC.cc ← 2#100#;  
elseif src1 = src2 then AC.cc ← 2#010#;  
elseif src1 > src2 then AC.cc ← 2#001#;  
else AC.cc ← 2#000#; # indicates one number is a NaN  
end if;
```

**cmpr, cmpri**

<b>Faults:</b>	STANDARD	Refer to the discussion of faults at the beginning of this chapter.
	Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.
	The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.	
	Floating Invalid Operation	One or more operands are an SNaN value.
<b>Example:</b>	<pre>cmprl g2, g6 # compare values in g6, g7               # and g2, g3</pre>	
<b>Opcode:</b>	<pre>cmpr      685 REG cmpri     695 REG</pre>	
<b>See Also:</b>	cmpr, cmpi, BRANCH IF	

**Action:**

```

AC.cc ← 2#010#;
for i in 0 .. len - 1 loop
  if byte (src1 + i) > byte (src2 + i)
    then AC.cc ← 2#001#;
  Exit;
elseif byte (src1 + i) < byte (src2 + i)
  then AC.cc ← 2#100#;
  Exit;
end if;
end loop;

```

**cmpstr****Mnemonic:** **cmpstr** Compare String

**Format:** **cmpstr** *src1*, *src2*, *len*  
                   reg          reg          reg/lit  
                   addr          addr

**Description:** Compares two strings of equal length and sets the condition code according to the result. The *src1* and *src2* operands specify the addresses of the first byte in each string, and the *len* operand specifies the string length, in bytes. The *len* operand can range from 0 to  $2^{32} - 1$ .

If the strings are identical, the condition code is set to  $010_2$ ; if they are not identical, the condition code is set to  $100_2$  or  $001_2$ , as explained in the next paragraph.

The two strings are compared in lexicographical order. This means that the processor compares the strings byte-by-byte according to their ASCII value. If the byte-by-byte comparison shows that the two strings are identical, the condition code is set to  $010_2$ . When two bytes of different ASCII value are found, the processor sets the condition code to  $001_2$  if the value of the byte from the *src1* string is greater than the value of the byte from the *src2* string or to  $100_2$  if the byte from the *src1* string is less than the byte from the *src2* string.

**Action:**

```

AC.cc ← 2#010#;
for i in 0 .. len - 1 loop
  if byte (src1 + i) > byte (src2 + i)
    then AC.cc ← 2#001#;
    Exit;
  elseif byte (src1 + i) < byte (src2 + i)
    then AC.cc ← 2#100#;
    Exit;
  end if;
end loop;
```

cmpstr

Faults:

STANDARD

Example:

cmpstr g3, g8, 25  
# compare strings that are 25 bytes long and  
# that begin at the addresses given in  
# registers g3 and g8

Opcode:

cmpstr 603 REG

See Also:

movstr, movqstr, fill

Format:

cmpstr\* src1, src2, targ  
reg/lit reg  
cmpstr\* src1, src2, targ  
reg/lit reg

Description:

Compares the src2 and src1 values and sets the condition code in the arithmetic controls according to the results of the comparison. If the logical AND of the condition code and the mask-part of the opcode is not zero, the processor branches to the instruction specified with the targ operand; otherwise, the processor goes to the next instruction. When using the Intel 80960MC Assembler, the targ operand must be a label, which specifies the IP of the target instruction.

The targ operand can be no farther than -2<sup>12</sup> to 2<sup>12</sup> - 4 bytes from the current IP.

NOTE

At the machine level, the compare-and-branch instructions use the COBR instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by displacement in the following action statement), which can range from -2<sup>10</sup> to 2<sup>10</sup> - 1. To determine the IP of the target instruction, the processor converts this displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

## COMPARE AND BRANCH

<b>Mnemonics:</b>	<b>cmpibe</b>	Compare Integer And Branch If Equal
	<b>cmpibne</b>	Compare Integer And Branch If Not Equal
	<b>cmpibl</b>	Compare Integer And Branch If Less
	<b>cmpible</b>	Compare Integer And Branch If Less Or Equal
	<b>cmpibg</b>	Compare Integer And Branch If Greater
	<b>cmpibge</b>	Compare Integer And Branch If Greater Or Equal
	<b>cmpibo</b>	Compare Integer And Branch If Ordered
	<b>cmpibno</b>	Compare Integer And Branch If Unordered
	<b>cmpobe</b>	Compare Ordinal And Branch If Equal
	<b>cmpobne</b>	Compare Ordinal And Branch If Not Equal
	<b>cmpobl</b>	Compare Ordinal And Branch If Less
	<b>cmpoble</b>	Compare Ordinal And Branch If Less Or Equal
	<b>cmpobg</b>	Compare Ordinal And Branch If Greater
	<b>cmpobge</b>	Compare Ordinal And Branch If Greater Or Equal

<b>Format:</b>	<b>cmpib*</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg	<i>targ</i> disp
	<b>cmpob*</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg	<i>targ</i> disp

**Description:** Compares the *src2* and *src1* values and sets the condition code in the arithmetic controls according to the results of the comparison. If the logical AND of the condition code and the mask-part of the opcode is not zero, the processor branches to the instruction specified with the *targ* operand; otherwise, the processor goes to the next instruction. When using the Intel 80960MC Assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

The *targ* operand can be no farther than  $-2^{12}$  to  $(2^{12} - 4)$  bytes from the current IP.

### NOTE

At the machine level, the compare-and-branch instructions use the COBR instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statement), which can range from  $-2^{10}$  to  $(2^{10} - 1)$ . To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

## COMPARE AND BRANCH

To allow labels to be used in the assembly-language versions of these instructions, the Intel 80960MC Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

$$\text{displacement} = (\text{targ} - \text{IP})/4$$

For further information about the COBR instruction format, refer to Appendix B.

The following table shows the condition-code mask for each instruction:

Instruction	Mask	Branch Condition
<b>cmpibno</b>	000	No Condition
<b>cmpibg</b>	001	$\text{src1} > \text{src2}$
<b>cmpibe</b>	010	$\text{src1} = \text{src2}$
<b>cmpibge</b>	011	$\text{src1} \geq \text{src2}$
<b>cmpibl</b>	100	$\text{src1} < \text{src2}$
<b>cmpibne</b>	101	$\text{src1} \neq \text{src2}$
<b>cmpible</b>	110	$\text{src1} \leq \text{src2}$
<b>cmpibo</b>	111	Any Condition
<b>cmpobg</b>	001	$\text{src1} > \text{src2}$
<b>cmpobe</b>	010	$\text{src1} = \text{src2}$
<b>cmpobge</b>	011	$\text{src1} \geq \text{src2}$
<b>cmpobl</b>	100	$\text{src1} < \text{src2}$
<b>cmpobne</b>	101	$\text{src1} \neq \text{src2}$
<b>cmpoble</b>	110	$\text{src1} \leq \text{src2}$

The **cmpibo** instruction always branches; the **cmpibno** instruction never branches.

The functions that these instructions perform can be duplicated with a **cmpi** instruction followed by a branch-if instruction, as described in the description of the **cmpi** instruction in this chapter.



## COMPARE AND BRANCH

**Action:** if  $src1 < src2$  then  $AC.cc \leftarrow 2\#100\#$ ;  
 elseif  $src1 = src2$  then  $AC.cc \leftarrow 2\#010\#$ ;  
 else  $AC.cc \leftarrow 2\#001\#$ ;  
 end if;  
 if mask and  $AC.cc \neq 2\#000\#$   
   then  $IP \leftarrow IP + 4 + (displacement * 4)$ ;  
   # resume execution at the new IP  
 else  $IP \leftarrow IP + 4$ ;  
 # resume execution at the next IP  
 end if;

**Faults:** STANDARD

**Example:** # assume  $g3 < g9$   
 cmpibl  $g3, g9, xyz$  #  $g9$  is compared with  $g3$ ;  
                           #  $IP \leftarrow xyz$ .  
 # assume  $r7 \geq 19$   
 cmpobge  $r7, 19, xyz$  #  $19$  is compared with  $r7$ ;  
                           #  $IP \leftarrow xyz$ .

**Opcode:**

cmpibe	3A	COBR
cmpibne	3D	COBR
cmpibl	3C	COBR
cmpible	3E	COBR
cmpibg	39	COBR
cmpibge	3B	COBR
cmpibo	3F	COBR
cmpibno	38	COBR
cmpobe	32	COBR
cmpobne	35	COBR
cmpobl	34	COBR
cmpoble	36	COBR
cmpobg	31	COBR
cmpobge	33	COBR

**See Also:** BRANCH IF, cmpi

**concmpi, concmpo**

**Mnemonics:**    **concmpi**    Conditional Compare Integer  
                   **concmpo**    Conditional Compare Ordinal

**Format:**        **concmp\***    *src1*,        *src2*  
     reg/lit        reg/lit

**Description:**    Compares the *src2* and *src1* values if bit 2 of the condition code is not set. If the comparison is performed, the condition code is set according to the results of the comparison.

These instructions are provided to facilitate bounds checking by means of two-sided range comparisons (e.g., is A between B and C?). They are generally used after a compare instruction to test whether a value is inclusively between two other values.

The example below illustrates this application by testing whether the value in g3 is between the values in g5 and g6, where g5 is assumed to be less than g6. First a comparison (**cmpo**) of g3 and g6 is performed. If g3 is less than or equal to g6 (i.e., condition code is either 010<sub>2</sub> or 001<sub>2</sub>), a conditional comparison (**concmpo**) of g3 and g5 is then performed. If g3 is greater than or equal to g5 (indicating that g3 is within the bounds of g5 and g6), the condition code is set to 010<sub>2</sub>; otherwise, it is set to 001<sub>2</sub>.

**Action:**            **if (AC.cc and 2#100#) = 0 then**  
                           **if *src1* ≤ *src2***  
                               **then AC.cc ← 2#010;**  
                               **else AC.cc ← 2#001;**  
                           **endif;**  
                   **endif;**

**Faults:**            STANDARD

**Example:**        *cmpo g6, g3*        # compares g6 and g3 and  
     # sets AC.cc  
                   *concmpo g5, g3*    # if AC.cc is not 1XX,  
     # g5 is compared with g3

**Opcode:**        **concmpi**    5A3        REG  
                       **concmpo**    5A2        REG

**See Also:**        *cmpo*, *cmpi*

condrec

**Mnemonic:** condrec Conditional Receive

**Format:** condrec src, dst  
reg reg  
SS SS

**Description:** Attempts to receive a message from a port and sets the condition code to indicate whether the message was received successfully or not. The *src* operand contains the SS of the port.

The processor must be in the supervisor mode to execute this instruction.

If the message is received successfully, the SS of the message is stored in the *dst* operand, the condition code is set to 010<sub>2</sub>, and execution of the process continues.

If a message is not available, the condition code is set to 000<sub>2</sub> and execution of the process continues.

This instruction is similar to the **receive** instruction, except that with the **receive** instruction, the process blocks and is suspended if a message is not available at the port.

**Action:**

```

x ← atomic_read(port.lock);
if least_significant_bit(x) = 1
then atomic_write(port.lock) ← x;
go to condrec;
else atomic_write(port.lock) ← x or 1;
if port.Q = 1 or port is empty
then AC.cc ← 2#000#;
else if port is fifo
then dequeue first message;
else dequeue first message from
highest-priority nonempty queue;
dst ← message_SS;
AC.cc ← 2#010#;
x ← atomic_read(port.lock);
atomic_write(port.lock) ← x xor 1;
endif;
```

condrec

**Faults:** STANDARD

**Example:** # Assume message is available at port  
condrec r8, r9  
# message SS from port specified in  
# r8 is stored in r9;  
# AC.cc is set to 2#010#

**Opcode:** condrec 646 REG

**See Also:** receive, send

The processor must be in the supervisor mode to execute this instruction. The processor checks the semaphore count and the semaphore queue tail. If the count is non-zero and the queue tail is zero, the count is decremented by one, the condition code is set to 010<sub>2</sub> (indicating a successful wait), and execution of the process continues.

If the count is zero or the queue tail is non-zero, the condition code is set to 000<sub>2</sub> (indicating an unsuccessful wait) and execution of the process continues.

This instruction is similar to the wait instruction, except that with the wait instruction, the process is suspended and enqueued on the semaphore if the semaphore count is zero or the semaphore queue tail is non-zero.

```

end if
atomic_write (semaphore.lock) ← x xor 1;
x ← atomic_read (semaphore.lock);
end if;
AC.cc ← 2#010#;
else semaphore.count ← semaphore.count - 1;
then AC.cc ← 2#000#;
if (semaphore.count = 0) or (semaphore.tail) ≠ 0)
else atomic_write (semaphore.lock) ← x or 1;
go to condwait;
then atomic_write (semaphore.lock) ← x;
if least_significant_bit(x) = 1
x ← atomic_read (semaphore.lock);

```

**Action:**

## condwait

**Mnemonics:** condwait Conditional Wait

**Format:** condwait *src*  
reg  
SS

**Description:** Attempts to wait on the semaphore and sets the condition code to indicate whether the wait was completed successfully or not. The *src* operand contains the SS of the semaphore.

The processor must be in the supervisor mode to execute this instruction.

The processor checks the semaphore count and the semaphore queue tail. If the count is non-zero and the queue tail is zero, the count is decremented by one, the condition code is set to 010<sub>2</sub> (indicating a successful wait), and execution of the process continues.

If the count is zero or the queue tail is non-zero, the condition code is set to 000<sub>2</sub> (indicating an unsuccessful wait) and execution of the process continues.

This instruction is similar to the **wait** instruction, except that with the **wait** instruction, the process is suspended and enqueued on the semaphore if the semaphore count is zero or the semaphore queue tail is non-zero.

**Action:**

```

x ← atomic_read (semaphore.lock);
if least_significant_bit(x) = 1
  then atomic_write (semaphore.lock) ← x;
  go to condwait;
else atomic_write (semaphore.lock) ← x or 1;
  if (semaphore.count = 0) or (semaphore.tail) ≠ 0)
    then AC.cc ← 2#000#;
    else semaphore.count ← semaphore.count - 1;
    AC.cc ← 2#010#;
  end if;
x ← atomic_read (semaphore.lock);
atomic_write (semaphore.lock) ← x xor 1;
end if

```

condwait

Faults: STANDARD

Example: # Assume semaphore count is non-zero and no  
# processes are queued at the semaphore.  
condwait g3  
# successful wait is performed on semaphore  
# specified with g3; AC.cc set to 2#010#

Opcode: condwait 668 REG

See Also: wait, signal

Src	Dst
-∞	*
-F	-1 to +1
-0	+1
+0	+1
+F	-1 to +1
+∞	*
NaN	NaN

Notes:  
F Means finite-real number  
\* Indicates floating invalid-operation exception

In the trigonometric instructions, the 80960MC uses a value for  $\pi$  with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 7 titled "PI" gives this  $\pi$  value, along with some suggestions for representing this value in a program.

Action: dst ← cosine (src);



**cosr, cosrl**

**Mnemonics:**    **cosr**            Cosine Real  
                  **cosrl**          Cosine Long Real

**Format:**        **cosr\***        *src*,        *dst*  
                                      freg/flit        freg

**Description:**    Calculates the cosine of the value in *src* and stores the result in *dst*. The *src* value is an angle given in radians. The resulting *dst* value is in the range -1 to +1, inclusive.

For the **cosrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the cosine of various classes of numbers with neither overflow nor underflow.

Src	Dst
$-\infty$	*
-F	-1 to +1
-0	+1
+0	+1
+F	-1 to +1
$+\infty$	*
NaN	NaN

Notes:

- F    Means finite-real number
- \*    Indicates floating invalid-operation exception

In the trigonometric instructions, the 80960MC uses a value for  $\pi$  with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 7 titled "Pi" gives this  $\pi$  value, along with some suggestions for representing this value in a program.

**Action:**             $dst \leftarrow \text{cosine}(src);$

cosr, cosrl

**Faults:** STANDARD Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Invalid Operation The *src* operand is  $\infty$ .  
One or more operands are an SNaN value.

Floating Inexact Result cannot be represented exactly in destination format.

**Example:** cosrl r8, g2 # cosine of value in r8, r9 is  
# stored in g2, g3

**Opcode:** cosr 68D REG  
cosrl 69D REG

**See Also:** sinr, sinrl, tanr, tanrl

**Faults:** STANDARD Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding One or more operands is a denormalized value and the normalizing-mode bit in the arithmetic controls is set.

**Example:** # absolute value from fp0 is copied to  
# fp2; sign from fp1 is copied to fp2  
cpxsr fp0, fp1, fp2

**Opcode:** cpxsr 6E3 REG  
cpxsr 6E3 REG

## cpysre, cpysre

**Mnemonics:** **cpysre** Copy Sign Real Extended  
**cpysre** Copy Reversed Sign Real Extended

**Format:** **cpy\*** *src1*, *src2*, *dst*  
 freg/flit freg/flit freg

**Description:** Copies the absolute value of *src1* into *dst*. For the **cpysre** instruction, the sign of *src2* is copied to *dst*; for the **cpysre** instruction, the opposite of the sign of *src2* is copied to *dst*.

If the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of three successive registers. Also, the number of this register must be a multiple of four (e.g., g0, g4, g8).

These instructions only operate on values in the extended-real format. The same operations can be performed on real- and long-real values using the **setbit** and **clearbit** instructions, or a combination of the **chkbit** and **alterbit** instructions.

**Action:**

**cpysre:** if *src2* is positive then *dst* ← abs(*src1*);  
 else *dst* ← -abs(*src1*);  
 endif;

**cpysre:** if *src2* is negative then *dst* ← abs(*src1*);  
 else *dst* ← -abs(*src1*);  
 endif;

**Faults:** STANDARD Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding One or more operands is a denormalized value and the normalizing-mode bit in the arithmetic controls is set.

**Example:**

```
cpysre fp0, fp1, fp2
# absolute value from fp0 is copied to
# fp2; sign from fp1 is copied to fp2
```

**Opcode:**

<b>cpysre</b>	6E2	REG
<b>cpysre</b>	6E3	REG

## cvtilr, cvtir

<b>Mnemonics:</b>	<b>cvtilr</b>	Convert Long Integer to Real	
	<b>cvtir</b>	Convert Integer to Real	
<b>Format:</b>	<b>cvti*</b>	<i>src</i> , reg/lit	<i>dst</i> freg
<b>Description:</b>	<p>Converts the integer in <i>src</i> to a real and stores the result in <i>dst</i>. For the <b>cvtilr</b> instruction, the <i>src</i> operand references the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).</p> <p>Converting an integer to long real format requires two instructions. First, the integer is converted to extended real format by using the <b>cvtir</b> or <b>cvtilr</b> instruction with a floating-point register as a destination. Then the <b>movrl</b> instruction is used to move the value from the floating-point register to two global or local registers, causing an explicit conversion to long real format. (Note that this conversion is always exact.) The example section below illustrates this conversion.</p>		
<b>Action:</b>	<i>dst</i> ← real ( <i>src</i> );		
<b>Faults:</b>	STANDARD	Refer to the discussion of faults at the beginning of this chapter.	
	<p>The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.</p>		
	Floating Inexact	Can only be signaled when converting an integer to real (32-bit) format	
<b>Example:</b>	<pre># Conversion of an integer to a long real value cvtir g6, fp3 movrl fp3, g8 # result stored in g8, g9</pre>		
<b>Opcode:</b>	<b>cvtir</b>	674	REG
	<b>cvtilr</b>	675	REG
<b>See Also:</b>	cvtri, movr		

cvtri, cvtril, cvtzri, cvtzril

Mnemonics:	cvtri	Convert Real To Integer
	cvtril	Convert Real To Integer Long
	cvtzri	Convert Truncated Real To Integer
	cvtzril	Convert Truncated Real To Long Integer

Format:	cvtri*	src, dst
		freg/flit reg

**Description:** Converts the real value in *src* to an integer and stores the result in *dst*.

For the **cvtril** and **cvtzril** instructions, the *dst* operand references the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The nontruncated versions of these instructions round according to the current rounding mode in the Arithmetic Controls register. The truncated versions always round toward zero.

Converting a long real value to an integer requires two instructions. First, the long real value is converted to extended real format by using the **movrl** instruction with a floating-point register as a destination. (Note that this operation is always exact.) Then one of the convert real-to-integer instructions is used to move the value from the floating-point register to one or two global or local registers. The example section below illustrates this conversion.

If the magnitude of the result cannot be represented in the destination, an integer-overflow fault is raised, and the maximum positive or maximum negative value is stored in the destination (depending on whether the real value was positive or negative, respectively).

**Action:** *dst* ← integer (*src*);  
# *src* is rounded to integer value

Opcode:	cvtri	674	RBC
	cvtril	675	RBC

See Also: **cvtri, movr**

**cvtri, cvtril, cvtzri, cvtzril**

**Faults:** STANDARD Refer to the discussion of faults at the beginning of this chapter.

The following exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls register.

Integer Overflow Result is too large for destination format.

**Example:** # Conversion of long real value to an integer  
 movrl g4, fp2 # long-real source is  
 # converted to extended-real  
 # format and moved to fp2  
 cvtril fp2, g12 # extended-real value is  
 # converted to long integer

**Opcode:** cvtri 6C0 REG  
 cvtril 6C1 REG  
 cvtzri 6C2 REG  
 cvtzril 6C3 REG

**See Also:** cvtir, movr



**daddc**

**Mnemonic:** **daddc** Decimal Add With Carry

**Format:** **daddc** *src1*, *src2*, *dst*  
reg reg reg

**Description:** Adds bits 0 through 3 of *src2* and *src1* and bit 1 of the condition code (used here as a carry bit). The result is stored in bits 0 through 3 of *dst*. If the addition results in a carry, bit 1 of the condition code is set. Bits 4 through 31 of *src2* are copied to *dst* unchanged.

This instruction is intended to be used iteratively to add binary-coded-decimal (BCD) values in which the least-significant four bits of the operands represent the decimal numbers 0 to 9. The instruction assumes that the least significant 4 bits of both operands are valid BCD numbers. If these bits are not valid BCD numbers, the resulting value in *dst* is unpredictable.

**Action:** # Let the value of the condition code be xCx.  
 $dst \leftarrow src2 + src1 + C;$   
 $AC.cc \leftarrow 2\#0C0\#;$   
# C is carry from addition of bits 0 through 3 of operands  
# Bits 4 - 31 of *dst* are same as bits 4 - 31 of *src2*

**Faults:** STANDARD

**Example:** `daddc g5, g9, g10` #  $g10 \leftarrow g9 + g5 + \text{Carry Bit}$   
# where arithmetic is  
# carried out only on bits 0  
# through 3 of the operands

**Opcode:** **daddc** 642 REG

**See Also:** **dsubc, dmovt**



**divr, divrl**

**Mnemonic:** **divr** Divide Real  
**divrl** Divide Long Real

**Format:** **divr\*** *src1*, *src2*, *dst*  
 freg/flit freg/flit freg

**Description:** Divides the *src2* value by the *src1* value and stores the result in *dst*.

For the **divrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the source values is 0,  $\infty$ , or a NaN.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						
Src2		$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
	$-\infty$	*	$+\infty$	$+\infty$	$-\infty$	$-\infty$	*	NaN
	-F	+0	+F	**	**	-F	-0	NaN
	-0	+0	+0	*	*	-0	-0	NaN
	+0	-0	-0	*	*	+0	+0	NaN
	+F	-0	-F	**	**	+F	+0	NaN
	$+\infty$	*	$-\infty$	$-\infty$	$+\infty$	$+\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

- F Means finite-real number.
- \* Indicates floating invalid-operation exception.
- \*\* Indicates floating zero-divide exception.

**Action:**  $dst \leftarrow src2 / src1;$

**divr, divrl**

<b>Faults:</b>	STANDARD	Refer to the discussion of faults at the beginning of this chapter.
	Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.
	The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.	
	Floating Overflow	Result is too large for destination format.
	Floating Underflow	Result is too small for destination format.
	Floating Zero Divide	The <i>src1</i> operand is 0 and the <i>src2</i> operand is numeric and finite.
	Floating Invalid Operation	Both source operands are 0 or both are $\infty$ . One or more operands are an SNaN value.
	Floating Inexact	Result cannot be represented exactly in destination format.

**Example:** `divrl g10, g0, fp1 # fp1 ← g0, g1 / g10, g11`

<b>Opcode:</b>	<b>divr</b>	78B	REG
	<b>divrl</b>	79B	REG

**See Also:** `ediv, mulr, mulrl`

**dmovt**

**Mnemonic:** **dmovt** Decimal Move And Test

**Format:** **dmovt** *src*, *dst*  
reg reg

**Description:** Copies the *src* value into *dst*. The least-significant eight bits of the *src* value are tested to determine whether or not they constitute a valid ASCII decimal (00110000<sub>2</sub> .. 00111001<sub>2</sub>), and the condition code is set accordingly. If the value is a valid ASCII decimal, the condition code is set to 000<sub>2</sub>; otherwise, it is set to 010<sub>2</sub>.

This instruction is intended to be used iteratively to validate decimal strings.

**Action:**  $dst \leftarrow src$ ;  
if  $src = 2\#0011000\# \dots 2\#00111001\#$   
then AC.cc  $\leftarrow 2\#000\#$ ;  
else AC.cc  $\leftarrow 2\#010\#$ ;  
end if;

**Faults:** STANDARD

**Example:** `dmovt g1, g6 # g6  $\leftarrow$  g1;  
# g1 tested for decimal value`

**Opcode:** **dmovt** 644 REG

**See Also:** **daddc, dsubc**

**dsubc**

**Mnemonic:** **dsubc** Decimal Subtract With Carry

**Format:** **dsubc** *src1*, *src2*, *dst*  
reg reg reg

**Description:** Subtracts bits 0 through 3 of *src2* and *src1* and bit 1 of the condition code (used here as a carry bit). The result is stored in bits 0 through 3 of *dst*. If the subtraction results in a carry, bit 1 of the condition code is set. Bits 4 through 31 of *src* are copied to *dst* unchanged.

This instruction is intended to be used iteratively to subtract binary-coded-decimal (BCD) values in which the least-significant four bits of the operands represent the decimal numbers 0 to 9. The instruction assumes that the least significant 4 bits of both operands are valid BCD numbers. If these bits are not valid BCD numbers, the resulting value in *dst* is unpredictable.

**Action:** # Let the value of the condition code be xCx.  
 $dst \leftarrow src2 - src1 - 1 + C;$   
 $AC.cc \leftarrow 2\#0C0\#;$   
 # C is carry from subtraction of bits 0 through 4 of operands  
 # Bits 4-31 of *dst* are same as bits 4-31 of *src2*

**Faults:** STANDARD

**Example:** `dsubc r1, r2, r12` #  $r12 \leftarrow r2 - r1 - 1 + \text{Carry}$   
 # Bit, where arithmetic is  
 # carried out only on bits 0  
 # through 3 of the operands

**Opcode:** **dsubc** 643 REG

**See Also:** `daddc`, `dmovt`



## ediv

**Mnemonic:** **ediv**

Extended Divide

**Format:** **ediv**

*src1*, *src2*, *dst*  
reg/lit reg/lit reg

**Description:** Divides *src2* by *src1* and stores the result in *dst*. The *src2* value is a long ordinal (i.e., 64 bits), which is contained in two adjacent registers. The *src2* operand specifies the lower numbered register, which contains the least significant bits of the operand. The *src2* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ...). The *src1* value is a normal ordinal (i.e., 32 bits).

The remainder is stored in the register designated by *dst* and the quotient is stored in the next highest numbered register. The *dst* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ...).

This instruction performs ordinal arithmetic.

If this operation overflows (i.e., the quotient or remainder do not fit in 32-bits), no fault is raised and the result is undefined.

**Action:**

$dst \leftarrow (src2 - (src2 / src1) * src1);$  # remainder  
 $dst + 1 \leftarrow (src2 / src1);$  # quotient

**Faults:**

STANDARD, Arithmetic Zero-Divide

**Example:**

`ediv g3, g4, g10` # g10 ← remainder of g4, g5/g3  
# g11 ← quotient of g4, g5/g3

**Opcode:** **ediv**

671 REG

**See Also:**

**emul**

## emul

**Mnemonic:** **emul** Extended Multiply

**Format:** **emul** *src1*, *src2*, *dst*  
reg/lit reg/lit reg

**Description:** Multiplies *src2* by *src1* and stores the result in *dst*. The result is a long ordinal (i.e., 64 bits), which is stored in two adjacent registers. The *dst* operand specifies the lower numbered register, which receives the least significant bits of the result. The *dst* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ...).

This instruction performs ordinal arithmetic.

**Action:**  $dst \leftarrow (src1 * src2) \bmod 2^{32};$   
 $dst + 1 \leftarrow (src * src2) / \bmod 2^{32};$

**Faults:** STANDARD

**Example:** `emul r4, r5, g2 # g2, g3 ← r4 * r5`

**Opcode:** **emul** 670 REG

**See Also:** **ediv**



**expr, expr1**

<b>Faults:</b>	STANDARD	Refer to the discussion of faults at the beginning of this chapter.
	Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.
	The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.	
	Floating Underflow	Result is too small for destination format.
	Floating Invalid Operation	One or more operands are an SNaN value.
	Floating Inexact	Result cannot be represented exactly in destination format.

**Example:**

```

# y = 2^x (y and x in g0)
# uses identity
# 2^x = 2^(I+f)
# = 2^I * ((2^f - 1) + 1)
# where: I integer, -0.5 <= f <= +0.5
# assumes round-to-nearest
# does not handle infinities or NaNs
_pow2x:
    roundr    g0,fp0          # I in fp0
    subr      fp0,g0,g0       # f in g0
    expr      g0,g0
    addr      0f1.0,g0,g0
    cvtri     fp0,g1
    scaler    g1,fp0,g0

```

**Opcode:**

<b>expr</b>	689	REG
<b>expr1</b>	699	REG

**See Also:** scaler, logr

## extract

**Mnemonic:** **extract** Extract

**Format:** **extract** *bitpos*, *len*, *src/dst*  
reg/lit reg/lit reg

**Description:** Shifts a specified bit field in *src/dst* right and fills the bits to the left of the shifted bit field with zeros. The *bitpos* value specifies the least significant bit of the bit field to be shifted, and the *len* value specifies the length of the bit field.

**Action:**  $src/dst \leftarrow (src/dst / 2^{(bitpos \bmod 32)})$   
and  $(2^{len} - 1)$ ;

**Faults:** STANDARD

**Example:** `extract 5, 12, g4 # g4 ← g4 with bits 5  
# through 16 shifted right`

**Opcode:** **extract** 651 REG

**See Also:** **modify**

## FAULT IF

<b>Mnemonic:</b>	<b>faulte</b>	Fault If Equal
	<b>faultne</b>	Fault If Not Equal
	<b>faultl</b>	Fault If Less
	<b>faultle</b>	Fault If Less Or Equal
	<b>faultg</b>	Fault If Greater
	<b>faultge</b>	Fault If Greater Or Equal
	<b>faulto</b>	Fault If Ordered
	<b>faultno</b>	Fault If Unordered

**Format:** fault\*

**Description:** Raises a constraint-range fault if the logical AND of the condition code and the mask-part of the opcode is not zero.

The following table shows the condition-code mask for each instruction:

Instruction	Mask	Condition
faultno	000	Unordered
faultg	001	Greater
faulte	010	Equal
faultge	011	Greater or equal
faultl	100	Less
faultne	101	Not equal
faultle	110	Less or equal
faulto	111	Ordered

For the **faultno** instruction (unordered), the fault is raised if the condition code is equal to 2#000#.

**Action:** For all instructions except **faultno**:

```
if (mask and AC.cc) ≠ 2#000#
    then raise constraint-range fault;
end if;
```

**faultno:**

```
if AC.cc = 2#000#
    then raise constraint-range fault;
end if;
```



FAULT IF

Faults: STANDARD, Constraint Range

Example: # assume AC.cc AND 110 ≠ 000  
faultle  
# Constraint Range Fault is generated

Opcode:	faulte	1A	CTRL	
	faultne	1D	CTRL	
	faultl	1C	CTRL	
	faultle	1E	CTRL	
	faultg	19	CTRL	
	faultge	1B	CTRL	
	faulto	1F	CTRL	
	faultno	18	CTRL	

See Also: be, teste

Instruction	Mask	Condition
faultno	000	Unordered
faultg	001	Greater
faulte	010	Equal
faultge	011	Greater or equal
faultl	100	Less
faultne	101	Not equal
faultle	110	Less or equal
faulto	111	Ordered

For the faultno instruction (unordered), the fault is raised if the condition code is equal to 2#000#.

Action: For all instructions except faultno:  
if (mask and AC.cc) ≠ 2#000#  
then raise constraint-range fault;  
end if;  
faultno:  
if AC.cc = 2#000#  
then raise constraint-range fault;  
end if;

fill

**Format:** fill dst value len  
reg reg/lit reg/lit  
addr

**Mnemonic:** fill Fill String

**Description:** Fills a string in memory with repeated copies of the word value given in value. The dst operand specifies the address of the first byte of the string, and the len operand specifies the length of the string in bytes.

**Action:** for i in 0 .. (len/4) - 1 loop  
word (dst + i) ← value;  
end loop;  
case len rem 4 is  
when 0: null;  
when 1: byte (dst + len - 1) ← value;  
when 2: halfword(dst + len - 2) ← value;  
when 3: halfword(dst + len - 3) ← value;  
byte (dst + len - 1) ← value/65536;  
end case;

**Faults:** STANDARD

**Example:** fill g2, g8, g3 # fills string beginning at  
# address g2 with word value  
# in g8; string length given  
# in g3

**Opcode:** fill 617 REG

**See Also:** cmpstr

## flushreg

**Mnemonic:** **flushreg** Flush Local Registers

**Format:** **flushreg**

**Description:** Copies the contents of all the cached local-register sets into their associated register-save areas in the procedure stack. The contents of all the local-register sets except for the current set are then marked as invalid. On a return, the local registers for the frame being returned to are then loaded from the stack.

This operation is also carried out when the save process (**saveprcs**) instruction is executed, although the **saveprcs** instruction also updates additional process specific information.

The **flushreg** instruction is provided to allow a compiler or applications program to circumvent the normal call/return mechanism of the processor. For example, a compiler may need to back up several frames in the stack on the next return, rather than using the normal return mechanism that returns one frame at a time. Here, the compiler uses the **flushreg** instruction to update the stack with the current states of the saved register sets. The compiler can then return to any frame in the stack without losing the contents of the saved local-register sets. To return to a frame other than the frame directly below the current frame, the compiler merely modifies the PFP in register r0 of the current frame to point to the frame that it wishes to return to.

**Action:** Each register set except the current set is flushed to its associated stack frame in memory and marked as purged, meaning that they will be reloaded from memory if and when they become the current local register set.

**Faults:** STANDARD

**Example:** **flushreg**

**Opcode:** **flushreg** 66D REG

**See Also:** **saveprcs**

**fmark**

**Mnemonic:** **fmark** Force Mark

**Format:** **fmark**

**Description:** Generates a breakpoint trace-event. This instruction causes a breakpoint trace-event to be generated, regardless of the setting of the breakpoint trace mode flag, providing the trace-enable bit (bit 0) of the process controls is set.

When a breakpoint trace event is detected, the trace-fault-pending flag (bit 10) of the process controls word and the breakpoint-trace-event flag (bit 23) of the trace controls are set. Before the next instruction is executed, a trace fault is generated.

For more information on trace-fault generation, refer to Chapter 12.

**Action:** if process.trace\_enable  
    then  
        raise trace breakpoint fault  
    endif

**Faults:** STANDARD, Breakpoint Trace

**Example:** ld xyz, r4  
addi r4, r5, r6  
fmark  
# Breakpoint trace event is generated at  
# this point in the instruction stream.

**Opcode:** **fmark** 66C REG

**See Also:** mark

inspace

Mnemonic:	inspace	Inspect Access			
Format:	inspace	src reg addr	dst reg		
Description:	Loads the effective page representation rights of the byte specified with <i>src</i> in <i>dst</i> . The <i>src</i> operand is an address contained in a register. The page representation rights are contained in a two-bit field (bits 1 and 2) in the page table entry for the page that contains the selected byte. This field is loaded into bits 0 and 1 of the <i>dst</i> .				
Action:	if segment descriptor invalid raise invalid-segment-descriptor fault else if offset > segment length raise segment-length fault else <i>dst</i> ← effective page-representation rights endif				
Faults:	STANDARD, Invalid Descriptor, Segment Length				
Example:	inspace g5 g9 # Loads page representation # rights of byte specified in g5 # into g9				
Opcode:	inspace	613	REG		

Mnemonic:	<b>ld</b>	Load	00	ld	OpCode
	<b>ldob</b>	Load Ordinal Byte	08	ldob	
	<b>ldos</b>	Load Ordinal Short	88	ldos	
	<b>ldib</b>	Load Integer Byte	C0	ldib	
	<b>ldis</b>	Load Integer Short	C8	ldis	
	<b>ldl</b>	Load Long	88	ldl	
	<b>ldt</b>	Load Triple	A0	ldt	
	<b>ldq</b>	Load Quad	B0	ldq	

Format:	<b>ld*</b>	<i>src</i> ,	<i>dst</i>	See Also:
		mem	reg	

**Description:** Copies a byte or string of bytes from memory into a register or group of successive registers. The *src* operand specifies the address of the first byte to be loaded. The full range of addressing modes may be used in specifying *src*. (Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.)

The *dst* operand specifies a register or the first (lowest numbered) register of successive registers.

The **ldob** and **ldib**, and **ldos** and **ldis** instructions load a byte and half word, respectively, and convert it to a full 32-bit word. The **ld**, **ldl**, **ldt**, and **ldq** instructions copy 4, 8, 12, and 16 bytes, respectively, from memory into successive registers.

For the **ldl** instruction, *dst* must specify an even numbered register (e.g., g0, g2, ..., g12). For the **ldt** and **ldq** instructions, *dst* must specify a register number that is a multiple of four (e.g., g0, g4, g8). If the data extends beyond register g15 or r15 for the **ldl**, **ldt**, or **ldq** instruction, the results are unpredictable.

**Action:**  $dst \leftarrow \text{memory}(src);$

**Faults:** STANDARD

**Example:** `ldl 2456 (r3), r10` # r10, r11 ← value of two  
# words beginning at offset  
# 2456 plus the address in  
# r3 in memory



# LOAD

Opcode:	ld	90	MEM	ld	Mnemonic:
	ldob	80	MEM	ldob	
	ldos	88	MEM	ldos	
	ldib	C0	MEM	ldib	
	ldis	C8	MEM	ldis	
	ldl	98	MEM	ldl	
	ldt	A0	MEM	ldt	
	ldq	B0	MEM	ldq	

See Also: MOVE, STORE

**Description:** Copies a byte or string of bytes from memory into a register or group of successive registers. The src operand specifies the address of the first byte to be loaded. The full range of addressing modes may be used in specifying src. (Refer to Chapter 2 for a complete discussion of the addressing modes available with memory-type operands.)

The dst operand specifies a register or the first (lowest numbered) register of successive registers.

The ldob and ldib, and ldos and ldis instructions load a byte and half word, respectively, and convert it to a full 32-bit word. The ld, lhl, lhl, and ldq instructions copy 4, 8, 12, and 16 bytes, respectively, from memory into successive registers.

For the ldl instruction, dst must specify an even numbered register (e.g., g0, g2, ..., g12). For the ldt and ldq instructions, dst must specify a register number that is a multiple of four (e.g., g0, g4, g8). If the data extends beyond register g12 or r12 for the ldl, ldt, or ldq instruction, the results are unpredictable.

**Action:** dst ← memory (src);

**Flags:** STANDARD

**Example:** ldl 2456 (r3), r10 # r10, r11 ← value of two # words beginning at offset # 2456 plus the address in # r3 in memory

## lda

**Mnemonic:** **lda** Load Address

**Format:** **lda** *src* *dst*  
                   mem reg  
                   efa

**Description:** Computes the effective address specified with *src* and stores it in *dst*. The *src* address is not checked for validity.

An important application of this instruction is to load a constant longer than 5 bits into a register. (To load a register with a constant of 5 bits or less, the move instruction (**mov**) can be used with a literal as the *src* operand.)

**Action:**  $dst \leftarrow efa(src);$

**Faults:** STANDARD

**Example:** `lda 58 (g9), g1` # Computes the effective  
                           # address specified with  
                           # 58 (g9) and stores it in g1  
                   `lda 0x749, r8` # loads the constant 0x749  
                           # in r8

**Opcode:** **lda** 8C MEM

**ldphy**

**Mnemonic:**    **ldphy**            Load Physical Address

**Format:**        **ldphy**        *src*,        *dst*  
                                      reg            reg  
                                      addr

**Description:**    Translates the address in *src* into a physical address and stores the result in *dst*. This instruction is provided to convert virtual addresses into physical addresses.

The address to be translated must reside in a register. The **lda** instruction can be used to compute an effective virtual address from an address specified with one of the processor's addressing modes. The **ldphy** instruction can then be used to translate this virtual address into a physical address.

**Action:**            *dst* ← physical address (*src*)

**Faults:**            STANDARD

**Example:**        `lda 58 (g9), g3`    # Computes the effective  
                                      # address specified with  
                                      # 58 (g9) and stores it in g3  
                                      `ldphy g3, r7`        # r7 ← physical address  
                                      # of address specified with g3

**Opcode:**        **ldphy**        614        REG

**See Also:**        **lda**



## logbnr, logbnrl

**Mnemonic:** **logbnr** Log Binary Real  
**logbnrl** Log Binary Long Real

**Format:** **logbnr\*** *src*, *dst*  
 freg/flit freg

**Description:** Calculates the  $\log_2$  (*src*) and stores the integral part of this value (i.e., the part to the left of the binary point) as a real number in *dst*. The result of this operation is an unbiased exponent. When *src* is a denormalized number, *dst* is the unbiased exponent that *src* would have if the format had unlimited exponent range.

(The fractional part of  $\log_2$  (*src*) is ignored. If the fractional part is needed, use the **logr** or **logrl** instruction.)

This instruction implements the IEEE recommended function *logb*. It is useful for calculating the order of magnitude of a number.

For the **logbnrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the log binary of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src	Dst
$-\infty$	$+\infty$
-F	$\pm F$
-0	**
+0	**
+F	$\pm F$
$+\infty$	$+\infty$
NaN	NaN

Notes:

F Means finite-real number  
 \*\* Indicates floating zero-divide exception

**logbnr, logbnrl**

Note that the significand of the *src* operand can be extracted by using the **scaler** or **scalerl** instruction.

**Action:**  $dst \leftarrow (\log_2(\text{unbiased exponent}(src)) - \text{fraction});$   
 # the integral part of the unbiased exponent of *src*  
 # is stored in *dst* as a biased real

**Faults:** STANDARD

Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

		Floating Underflow	Result is too small for destination format.
		Floating Invalid Operation	One or more operands are an SNaN value.
		Floating Inexact	Result cannot be represented exactly in destination format.
		Floating Zero Divide	The <i>src</i> operand is 0.
<b>Example:</b>	logbnrl g12, fp3	# fp3 ← integral part # of log <sub>2</sub> (g12, g13)	
<b>Opcode:</b>	logbnr 68A	REG	
	logbnrl 69A	REG	

**See Also:** logr, scaler



logepr, logepri

Mnemonic: logepr Log Epsilon Real  
logepri Log Epsilon Long Real

Format: logepr\* src1, src2, dst  
freg/flit freg/flit freg

Description: Calculates (src2 \* log<sub>2</sub> (src1 + 1)), and stores the result in dst.

For the logepri instruction, if the src1, src2, or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1			
		(1/√2) - 1 to -0	-0	+ 0	+ 0 to √2 - 1
Src2	-∞	-∞	*	*	-∞
	-F	+F	+ 0	-0	-F
	-0	+ 0	+ 0	-0	-0
	+ 0	-0	-0	+ 0	+ 0
	+ F	-F	-0	+ 0	+ F
	+ ∞	+ ∞	*	*	+ ∞
	NaN	NaN	NaN	NaN	NaN

Notes:

- F Means finite-real number.
- \* Indicates floating invalid-operation exception.

This instruction offers optimal accuracy for values of src1 + 1 close to 1 (i.e., for values of src1 close to 0). This expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in src2.

## logepr, logeprl

The following equation is used to calculate the scale factor for a particular logarithm base, where  $n$  is the logarithm base desired for the result stored in *dst*:

$$\text{scale factor} = \log_n 2$$

The range of *src1* is restricted to the following:

$$1/\sqrt{2} \leq \text{src1} + 1 \leq \sqrt{2}$$

When the *src1* operand is outside this range, the **logr** or **logrl** instruction can be used with very insignificant loss of accuracy by adding 1.0 to *src1*.

**Action:**  $\text{dst} \leftarrow \text{src2} * \log_2(\text{src1} + 1);$

### Faults:

STANDARD

Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow

Result is too large for destination format.

Floating Underflow

Result is too small for destination format.

Floating Invalid Operation

The *src1* operand is 0 and the *src2* operand is  $\infty$ .

The *src1* operand does not fall within the range defined in the above description section.

One or more operands are an SNaN value.

Floating Inexact

Result cannot be represented exactly in destination format.

# logopr, logeprl

**Example:** `logopr g8,g4,fp2`  
`#fp2 ← g4,g5 * log2(g8,g9 + 1)`

<b>Opcode:</b>	<b>logopr</b>	681	REG
	<b>logeprl</b>	691	REG

**See Also:** `logr`

The range of `src1` is restricted to the following:  

$$1 \leq \text{src1} \leq 2^{\text{src2}}$$
  
 When the `src1` operand is outside this range, the `logr` or `logeprl` instruction can be used with very insignificant loss of accuracy by adding 1.0 to `src1`.

**Action:** `dst ← src2 * log2(src1 + 1);`

**Flags:** STANDARD

Refer to the discussion of flags at the beginning of this chapter.  
 One or more operands is an unnormalized (including denormalized) value and the normalizing mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- Floating Overflow**  
Result is too large for destination format.
- Floating Underflow**  
Result is too small for destination format.
- Floating Invalid Operation**  
The `src1` operand is 0 and the `src2` operand is  $\infty$ .
- Floating Inexact**  
The `src1` operand does not fall within the range defined in the above description section.
- Floating Invalid Operation**  
One or more operands are an NaN value.
- Floating Inexact**  
Result cannot be represented exactly in destination format.

## logr, logrl

**Mnemonic:** **logr** Log Real  
**logrl** Log Long Real

**Format:** **logr\*** *src1*, *src2*, *dst*  
 freg/flit freg/flit freg

**Description:** Calculates ( $src2 * \log_2(src1)$ ), and stores the result in *dst*. (The **logbnr** and **logbnrl** instructions perform this function more efficiently, if only an estimate is needed.)

For the **logrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						
Src2		$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
	$-\infty$	*	*	**	**	$\pm\infty$	$-\infty$	NaN
	-F	*	*	**	**	$\pm F$	$-\infty$	NaN
	-0	*	*	*	*	$\pm 0$	*	NaN
	+0	*	*	*	*	$\pm 0$	*	NaN
	+F	*	*	**	**	$\pm F$	$+\infty$	NaN
	$+\infty$	*	*	**	**	$\pm\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

- F Means finite-real number.
- \* Indicates floating invalid-operation exception.
- \*\* Indicates floating zero-divide exception.

The **logr** instruction combined with the **expr** instruction forms the basis for the power function  $x^y$ .

## logr, logrl

Adding 1.0 to a number to be used as the *src1* operand will cause information to be lost. To perform this function, use the **logepr** or **logeprl** instruction.

These instructions provide a simple method of converting the result of the  $\log_2$  arithmetic into a value in another logarithm base by including a scale factor in *src2*. The following equation is used to calculate the scale factor for a particular logarithm base, where *n* is the logarithm base desired for the result stored in *dst*:

$$\text{scale factor} = \log_n 2$$

**Action:**  $\text{dst} \leftarrow \text{src2} * \log_2(\text{src1});$

**Faults:** STANDARD

Refer to the discussion of faults at the beginning of this chapter.

### Floating Reserved Encoding

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

NaN	++	+	0	-
NaN	∞	∞	∞	∞
NaN	∞	∞	∞	∞
NaN	*	*	*	*
NaN	*	*	*	*
NaN	∞	∞	**	**
NaN	∞	∞	**	**
NaN	∞	∞	**	**
NaN	∞	∞	**	**

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow

Result is too large for destination format.

Floating Underflow

Result is too small for destination format.

Floating Zero Divide

The *src1* operand is 0 and *src2* is non-zero.

Floating Invalid Operation

The *src1* and *src2* operands are both 0.

The *src1* operand is  $\infty$  and the *src2* operand is 0.

The *src1* operand is 1 and the *src2* operand is  $\infty$ .

The *src1* operand is negative and non-zero.

One or more operands are an SNaN value.

Floating Inexact

Result cannot be represented exactly in destination format.

logr, logrl

Example:		logrl r2, g8, g2 # g2,g3 ← g8,g9 * log2(r2,r3)	mark	mark	Mnemonic:
Opcode:		logr 682 logrl 692	REG REG	mark	Description:
See Also:		expr, logepr			
When a breakpoint trace event is detected, the trace-fault-pending flag (bit 10) of the process controls and the breakpoint-trace-event flag (bit 23) of the trace controls are set. Before the next instruction is executed, a trace fault is generated.					
If the breakpoint-trace mode has not been enabled, the mark instruction behaves like a no-op.					
For more information on trace-fault generation, refer to Chapter 12.					
Action:		if process.trace_enable and breakpoint_trace_flag then raise trace breakpoint fault endif			
Faults:		STANDARD, Breakpoint Trace			
Example:		# Assume that the breakpoint trace mode is enabled. ld xyz, r4 addi r4, r5, r6 mark # Breakpoint trace event is generated at # this point in the instruction stream.			
Opcode:		68B	mark	REG	
See Also:		fmark, modpc, modtc			



**mark**

**Mnemonic:**    **mark**            **Mark**

**Format:**        **mark**

**Description:**    Generates a breakpoint trace event if the breakpoint trace mode has been enabled. The breakpoint trace mode is enabled if the trace-enable bit (bit 0) of the process controls and the breakpoint-trace mode bit (bit 7) of the trace controls have been set.

When a breakpoint trace event is detected, the trace-fault-pending flag (bit 10) of the process controls and the breakpoint-trace-event flag (bit 23) of the trace controls are set. Before the next instruction is executed, a trace fault is generated.

If the breakpoint-trace mode has not been enabled, the **mark** instruction behaves like a no-op.

For more information on trace-fault generation, refer to Chapter 12.

**Action:**            **if** process.trace\_enable **and** breakpoint\_trace\_flag  
                          **then**  
                              raise trace breakpoint fault  
                          **endif**

**Faults:**            STANDARD, Breakpoint Trace

**Example:**        # Assume that the breakpoint trace mode is  
                      # enabled.  
                      ld xyz, r4  
                      addi r4, r5, r6  
                      mark  
                      # Breakpoint trace event is generated at  
                      # this point in the instruction stream.

**Opcode:**        **mark**        66B        REG

**See Also:**        fmark, modpc, modtc

**modac**

**Mnemonic:** modac Modify AC

<b>Format:</b>	<b>modac</b>	<i>mask</i> , reg/lit	<i>src</i> , reg/lit	<i>dst</i> , reg	<i>ibom</i>	<i>Format:</i>
----------------	--------------	--------------------------	-------------------------	---------------------	-------------	----------------

**Description:** Reads and modifies the arithmetic controls for the current process. The processor changes its internally cached arithmetic controls as specified with *mask* and *src*. The *src* operand contains the value to be placed in the arithmetic controls and the *mask* operand specifies the bits that may be changed. Only the bits set in *mask* are modified in the arithmetic controls. Once the arithmetic controls have been changed, their initial state is copied into *dst*.

This instruction only affects the arithmetic controls cached in the processor. The arithmetic controls in the PCB for the current process are not affected.

```

Action:      temp ← AC
                AC ← (src and mask) or
                  (AC and not (mask));
                dst ← temp;

```

**Faults:** STANDARD

```
Example:    modac g1, g9, g12 # AC ← g9, masked by g1
```

<b>Opcode:</b>	<b>modac</b>	<b>645</b>	<b>REG</b>
----------------	--------------	------------	------------

**See Also:** `modpc`, `modtc`



modify

Mnemonic:      **modify**      Modify

Format:          **modify**      *mask*,      *src*,      *src/dst*  
                                     reg/lit      reg/lit      reg

**Description:**      Modifies selected bits in *src/dst* with bits from *src*. The *mask* operand selects the bits to be modified: only the bits set in the *mask* operand are modified in *src/dst*.

**Action:**              *src/dst* ← (*src* and *mask*) or (*src/dst* and not (*mask*));

**Faults:**              STANDARD

**Example:**            modify g8, g10, r4      # r4 ← g10 masked by g8

**Opcode:**            **modify**      650              REG

**See Also:**            alterbit, extract

**Action:**

```
if mask ≠ 0
then if process.process_controls.execution_mode ≠ supervisor
then raise type-mismatch fault;
end if;
temp ← process.process_controls;
process.process_controls ←
(mask and src/dst) or
(process.process_controls and not (mask));
src/dst ← temp;
if temp.priority > process.process_controls.priority
then check_pending_interrupt;
# if continue here, no interrupt to do
end if;
else src/dst ← process.process_controls;
end if;
```

## modpc

**Mnemonic:** **modpc** Modify Process Controls

**Format:** **modpc** *src*, *mask*, *src/dst*  
reg/lit reg/lit reg

**Description:** Reads and modifies the process controls for the current process. The processor changes its internally cached process controls as specified with *mask* and *src/dst*. The *src/dst* operand contains the value to be placed in the process controls and the *mask* operand specifies the bits that may be changed. Only the bits set in the mask are modified in the process controls. Once the process controls have been changed, their initial value is copied into *src/dst*. The *src* operand is a dummy operand that should be set equal to the *mask* operand.

The processor must be in the supervisor mode to modify the process controls using this instruction. If the *mask* operand is set to 0, this instruction can be used to read the process controls, without the processor being in the supervisor mode.

This instruction only affects the process controls cached in processor. The process controls in the PCB for the current process are not affected. If the action of this instruction results in the priority of the current process being lowered, the interrupt table and dispatch port are checked.

Changing the state, resume, internal state, and trace enable fields of the process controls can lead to unpredictable behavior, as described in Chapter 13 in the section titled "Changing the Process-Controls Word."

**Action:**

```

if mask ≠ 0
  then if process.process_controls.execution_mode ≠ supervisor
    then raise type-mismatch fault;
  end if;
  temp ← process.process_controls;
  process.process_controls ←
    (mask and src/dst) or
    (process.process_controls and not (mask));
  src/dst ← temp;
  if temp.priority > process.process_controls.priority
    then check_pending_interrupts;
    # if continue here, no interrupt to do
  end if;
  else src/dst ← process.process_controls;
end if;

```

modpc

Faults: STANDARD, Type Mismatch

Example: modpc g9, g9, g8 # process controls ← g8  
# masked by g9

Opcode: modpc 655 REG

See Also: modac, modtc

Description: Reads and modifies the trace controls for the current process. The process changes its internally cached trace controls as specified with mask and src. The src operand contains the value to be placed in the trace controls and the mask operand specifies the bits that may be changed. Once the trace controls have been changed, their initial state is copied into dst.

This instruction only affects the trace controls cached in processor. The trace controls in the PCB for the current process are not affected.

Since bits 8 through 15 and 24 through 31 of the trace-controls word are reserved, the mask operand is ANDed with 00F00FF<sub>16</sub> to insure that these bits are not set in the mask.

The changed trace controls take effect on the first non-branching instruction fetched from memory. Since instructions are prefetched four at a time, the trace controls may not take effect for up to the next four instructions executed.

For more information on the trace controls, refer to Chapters 12 and 16.

Action: temp ← process.trace\_controls;  
temp1 ← 16#00F00FF# and mask;  
process.trace\_controls ←  
(temp1 and src) or  
(process.trace\_controls and not(temp1));  
dst ← temp;

Faults: STANDARD

Example: modtc g12, g10, g2  
# trace controls ← g10 masked by g12;  
# previous trace controls stored in g2

Opcode: modtc 654 REG

See Also: modac, modpc



## modtc

**Mnemonic:**    **modtc**        Modify Trace Controls

**Format:**        **modtc**    *mask*,    *src*,        *dst*  
                                  reg/lit    reg/lit        reg

**Description:**    Reads and modifies the trace controls for the current process. The processor changes its internally cached trace controls as specified with *mask* and *src*. The *src* operand contains the value to be placed in the trace controls and the *mask* operand specifies the bits that may be changed. Only the bits set in the mask are modified in the trace controls. Once the trace controls have been changed, their initial state is copied into *dst*.

This instruction only affects the trace controls cached in processor. The trace controls in the PCB for the current process are not affected.

Since bits 8 through 15 and 24 through 31 of the trace-controls word are reserved, the *mask* operand is ANDed with 00FF00FF<sub>16</sub> to insure that these bits are not set in the mask.

The changed trace controls take effect on the first non-branching instruction fetched from memory. Since instructions are prefetched four at a time, the trace controls may not take effect for up to the next four instructions executed.

For more information on the trace controls, refer to Chapters 12 and 16.

**Action:**            temp ← process.trace\_controls;  
                         temp1 ← 16#00FF00FF# **and** *mask*;  
                         process.trace\_controls ←  
                                  (temp1 **and** *src*) **or**  
                                  (process.trace\_controls **and** **not**(temp1));  
                         *dst* ← temp;

**Faults:**            STANDARD

**Example:**            modtc g12, g10, g2  
                         # trace controls ← g10 masked by g12;  
                         # previous trace controls stored in g2

**Opcode:**           **modtc**        654            REG

**See Also:**          modac, modpc

MOVE

Mnemonic:	mov	Move
	movl	Move Long
	movt	Move Triple
	movq	Move Quad
Format:	mov*	src, dst reg/lit reg
Description:	<p>Copies the content of one or more source registers (specified with the <i>src</i> operand) to one or more destination registers (specified with the <i>dst</i> operand).</p> <p>For the <b>movl</b>, <b>movt</b>, and <b>movq</b> instructions, the <i>src</i> and <i>dst</i> operands specify the first (lowest numbered) register of several successive registers. The <i>src</i> and <i>dst</i> registers must be even numbered (e.g., g0, g2) for the <b>movl</b> instruction and an integral multiple of four (e.g., g0, g4) for the <b>movt</b> and <b>movq</b> instructions.</p> <p>When the <i>src</i> and <i>dst</i> operands overlap, the value moved is unpredictable.</p>	
Action:	$dst \leftarrow src;$	
Faults:	STANDARD	
Example:	movt g8, r4 # r4, r5, r6 ← g8, g9, g10	
Opcode:	mov	5CC REG
	movl	5DC REG
	movt	5EC REG
	movq	5FC REG
See Also:	ld, movr, st	

**movqstr**

**Mnemonic:** **movqstr** Move Quick String

**Format:** **movqstr** *dst*, *src*, *len*  
                   reg      reg      reg/lit  
                   addr      addr

**Description:** Copies a string of bytes from one location in memory to another, where the source and destination strings are assumed not to overlap. The *src* operand specifies the address of the first byte of the source string and the *dst* operand specifies the address of the first byte of the destination string. The *len* operand specifies the length of the string in bytes and can range from 1 to  $2^{32}-1$ .

The *src* operand and the *dst* operand each specify a register, which contains an address.

If the strings overlap, the value copied is not predictable. (Use the **movstr** instruction instead.)

**Action:**           **for** *i* in 0 .. *len* - 1 **loop**  
                       byte (*dst* + *i*)  $\leftarrow$  byte (*src* + *i*);  
                       **end loop**;

**Faults:**           STANDARD

**Example:**       **movqstr** r9, r2 ,r12   # Copies string beginning  
   # at r2, which is  
   # r12 bytes long, to  
   # string beginning at r9

**Opcode:**       **movqstr**   604       REG

**See Also:**       **cmpstr**, **fill**, **movstr**

**movr, movre, movrl**

**Mnemonic:**    **movr**        Move Real  
                 **movrl**      Move Long Real  
                 **movre**      Move Extended Real

**Format:**       **movr\***    *src*,        *dst*  
                              freg/flit    freg

**Description:** Copies a real value from one or more source registers (specified with the *src* operand) to one or more destination registers (specified with the *dst* operand).

For the **movrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. For the **movre** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of three successive registers.

When copying real numbers between global or local registers and floating-point registers, conversion between real or long-real format to extended-real format is performed implicitly. Conversion between real and long-real formats must be done through floating-point registers and requires two instructions, as illustrated in the example below.

When the **movre** instruction moves an operand from global or local registers to a floating-point register, it automatically truncates the most-significant 16 bits of the word in the third register (refer to Figure 7-5). Likewise, when this instruction is used to move an operand from a floating-point register to global or local registers, it adds 16 zeros to the third word. The **movre** instruction is not a numeric instruction; it merely manipulates bits.

The **movr** and **movrl** instructions can cause a floating-point exception to be raised, which might result in a fault being raised, as is explained in the section below on faults. The **movre** instruction can never raise an exception and thus never faults.

**Action:**         $dst \leftarrow src;$

## movr, movre, movrl

<b>Faults:</b>	STANDARD	Refer to the discussion of faults at the beginning of this chapter.
	Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.
	The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.	
	Floating Overflow	Result is too large for destination format.
	Floating Underflow	Result is too small for destination format.
	Floating Invalid Operation	Source operand is an SNaN value.
	Floating Inexact	Result cannot be represented exactly in destination format.

**Example:**

```
# Conversion of real value in g3
# to a long real value, which is
# stored in g4,g5
movr g3, fp2
movrl fp2, g4
```

**Opcode:**

<b>movr</b>	6C9	REG
<b>movrl</b>	6D9	REG
<b>movre</b>	6E9	REG

**See Also:** **mov**

**movstr**

**Mnemonic:** **movstr** Move String

**Format:** **movstr** *dst*, *src*, *len*  
reg reg reg/lit  
addr addr

**Description:** Copies a string of bytes from one location in memory to another. The *src* operand specifies the address of the first byte of the source string and the *dst* operand specifies the address of the first byte of the destination string. The *len* operand specifies the length of the string in bytes and can range from 1 to  $2^{32}-1$ .

The *src* operand and the *dst* operand each specify a register, which contains an address.

If the strings overlap, the **movstr** algorithm guarantees that no byte of the source string is overwritten before it is copied into the destination string. If it is guaranteed that there are no overlaps, the **movqstr** instruction performs this operation faster.

**Action:**

```

if src ≤ dst
  then
    for i in 1 .. len loop
      byte (dst + len - i)
        ← byte (src + len - i);
    end loop;
  else
    for i in 0 .. len - 1 loop
      byte (dst + i) ← byte (src + i);
    end loop;
  end if;

```

**Faults:** STANDARD

**Example:**

```

movstr g5, g1, g9
# Copies string, which is g9 bytes long and
# begins at address g1, to address g5

```

**Opcode:** **movstr** 605 REG

**See Also:** **cmpstr**, **fill**, **movqstr**



mul, mulo

Mnemonic:	mul mulo	Multiply Integer Multiply Ordinal	
Format:	mul*	src1, reg/lit	src2, reg/lit
Description:	Multiplies the src2 value by the src1 value and stores the result in dst.		
Action:	dst ← src2 * src1;		
Faults:	STANDARD, Integer Overflow		
Example:	mul r3, r4, r9 # r9 ← r4 TIMES r3		
Opcode:	mul mulo	741 701	REG REG
See Also:	emul, mulr		

mulr, mulrl

Mnemonic: **mulr** Multiply Real  
**mulrl** Multiply Long Real

Format: **mulr\*** *src1*, *src2*, *dst*  
freg/flit freg/flit freg

Description: Multiplies the *src2* value by the *src1* value and stores the result in *dst*.

For the **mulrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the source values is 0,  $\infty$ , or a NaN.

The following table shows the results obtained when multiplying various classes of numbers together, assuming that neither overflow nor underflow occurs.

		Src1						
Src2		$-\infty$	<b>-F</b>	<b>-0</b>	<b>+0</b>	<b>+F</b>	$+\infty$	NaN
	$-\infty$	$+\infty$	$+\infty$	*	*	$-\infty$	$-\infty$	NaN
	<b>-F</b>	$+\infty$	<b>+F</b>	<b>+0</b>	<b>-0</b>	<b>-F</b>	$-\infty$	NaN
	<b>-0</b>	*	<b>+0</b>	<b>+0</b>	<b>-0</b>	<b>-0</b>	*	NaN
	<b>+0</b>	*	<b>-0</b>	<b>-0</b>	<b>+0</b>	<b>+0</b>	*	NaN
	<b>+F</b>	$-\infty$	<b>-F</b>	<b>-0</b>	<b>+0</b>	<b>+F</b>	$+\infty$	NaN
	$+\infty$	$-\infty$	$-\infty$	*	*	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

- F Means finite-real number.
  - \*
- Indicates floating invalid-operation exception.

When you need to multiply by the power of 2, the **scaler** and **scalerl** instructions can also be used.

**mulr, mulri**

**Action:**  $dst \leftarrow src2 * src1;$

**Faults:** STANDARD

Refer to the discussion of faults at the beginning of this chapter.

### Floating Reserved Encoding

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

## Floating Overflow

Result is too large for destination format.

## Floating Underflow

Result is too small for destination format.

### Floating Invalid Operation

One source operand is 0 and the other is  $\infty$

One or more operands are an SNaN value.

## Floating Inexact

Result cannot be represented exactly in destination format.

**Example:** `mulr1 g12, g4, fp2`    #  $fp2 \leftarrow g4, g5 * g12, g13$

<b>Opcode:</b>	<b>* mulr</b>	78C	REG
	<b>* mulrl</b>	79C	REG

**See Also:** `emul`, `muli`, `scaler`

nand

Mnemonic:	nand	Nand				
Format:	nand	src1, reg/lit	src2, reg/lit	dst, reg		
Description:	Performs a bitwise NAND operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .					
Action:	$dst \leftarrow (\text{not } (src2)) \text{ or } (\text{not } (src1));$					
Faults:	STANDARD					
Example:	nand g5, r3, r7 # r7					
Opcode:	nand	58E	REG			
See Also:	and, andnot, nor, not, notand, notor, or, ornot, xnor, xor					

nor

Mnemonic:	nor	Nor				
Format:	nor	src1, reg/lit	src2, reg/lit	dst reg		
Description:	Performs a bitwise NOR operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .					
Action:	$dst \leftarrow \text{not } (src2) \text{ and not } (src1);$					
Faults:	STANDARD					
Example:	nor g8, 28, r5 # r5 ← 28 NOR g8					
Opcode:	nor	588	REG			
See Also:	and, andnot, nand, not, notand, notor, or, ornot, xnor, xor					

## not, notand

<b>Mnemonic:</b>	<b>not</b>	Not		
	<b>notand</b>	Not And		
<b>Format:</b>	<b>not</b>	<i>src</i> , reg/lit	<i>dst</i> reg	
	<b>notand</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Performs a bitwise NOT ( <b>not</b> instruction) or NOT AND ( <b>notand</b> instruction) operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
<b>Action:</b>	<b>not:</b>	$dst \leftarrow \text{not} (src);$		
	<b>notand:</b>	$dst \leftarrow (\text{not} (src2)) \text{ and } src1;$		
<b>Faults:</b>	STANDARD			
<b>Example:</b>	<pre>not g2, g4      # g4 ← NOT g2 notand r5, r6, r7  # r7 ← NOT r6 AND r5</pre>			
<b>Opcode:</b>	<b>not</b>	58A	REG	
	<b>notand</b>	584	REG	
<b>See Also:</b>	and, andnot, nand, nor, notor, or, ornot, xnor, xor			



# notbit

<b>Mnemonic:</b>	notbit	Not Bit			
<b>Format:</b>	notbit	bitpos, src, dst reg/lit reg/lit reg			
<b>Description:</b>	Copies the <i>src</i> value to <i>dst</i> with one bit toggled. The <i>bitpos</i> operand specifies the bit to be toggled.				
<b>Action:</b>	$dst \leftarrow src \text{ xor } 2^{(bitpos \bmod 32)}$				
<b>Faults:</b>	STANDARD				
<b>Example:</b>	notbit r3, r12, r7	# r7 ← r12 with the bit specified in r3 toggled			
<b>Opcode:</b>	notbit	580	REG		
<b>See Also:</b>	alterbit, chkbit, clrbit, setbit				

notor

Mnemonic:	notor	Not Or				
Format:	notor	src1, reg/lit	src2, reg/lit	dst reg		
Description:	Performs a bitwise NOT OR operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .					
Action:	$dst \leftarrow (\text{not } (src2)) \text{ or } src1;$					
Faults:	STANDARD					
Example:	notor	g12, g3, g6	# g6	← NOT g3 OR g12		
Opcode:	notor	58D	REG			
See Also:	and, andnot, nand, nor, not, notand, or, ornot, xnor, xor					

## or, ornot

<b>Mnemonic:</b>	<b>or</b> <b>ornot</b>	<b>Or</b> <b>Or Not</b>	<b>Mnemonic:</b>	<b>notor</b> <b>Not Or</b>
<b>Format:</b>	<b>or</b> <i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<b>Format:</b>	<b>notor</b> <i>src1</i> , reg
<b>Description:</b>	<b>ornot</b> <i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<b>Description:</b>	Performs a bitwise NOT OR operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .
<b>Description:</b>	Performs a bitwise OR ( <b>or</b> instruction) or ORNOT ( <b>ornot</b> instruction) operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
<b>Action:</b>	<b>or:</b> $dst \leftarrow src2 \text{ or } src1$ ; <b>ornot:</b> $dst \leftarrow src2 \text{ or not } (src1)$ ;			
<b>Faults:</b>	STANDARD			
<b>Example:</b>	<pre>or 14, g9, g3      # g3 ← g9 OR 14 ornot r3, r8, r11   # r11 ← r8 OR NOT r3</pre>			
<b>Opcode:</b>	<b>or</b> 587	REG	<b>Opcode:</b>	<b>notor</b> 58D
	<b>ornot</b> 58B	REG		
<b>See Also:</b>	and, andnot, nand, nor, not, notand, notor, xnor, xor			

receive

receive

**Mnemonic:** receive Receive

**Format:** receive *src* *dst*  
reg reg  
SS SS

**Description:** Attempts to receive a message from a communications port. The *src* operand contains the SS of the port. If the port has enqueued messages, the SS of the message at the head of the message queue is stored in *dst* and execution continues.

The processor must be in the supervisor mode to execute this instruction.

If the port is empty (i.e., has no messages queued), the process is suspended, with its IP left pointing to the current instruction. The process is then enqueued at the port at the tail of the blocked-processes queue.

The receive-blocked process remains blocked until it reaches the head of the blocked-processes queue and a message is received at the port. This message is then stored in the PCB of the blocked process, and the process is dequeued from the communications port and enqueued to its dispatching port.

When the process is again dispatched, the processor resumes the **receive** instruction, but this time it reads the message stored in its PCB, rather than going to the communication port again.

**Action:**  $x \leftarrow \text{atomic\_read}(\text{port.lock});$   
**if** least\_significant\_bit( $x$ ) = 1  
     **then** atomic\_write(port.lock)  $\leftarrow x$ ;  
     **go to** receive;  
**else** atomic\_write(port.lock)  $\leftarrow x$  or 1;  
     **if** port.Q = 1 **or** port is empty  
         **then if** port is fifo  
             **then** enqueue process on port  
             port.queue\_tail\_SS  $\leftarrow$  process\_SS;

receive

receive

```
else enqueue process on port.queue(process.priority);
    port.queue_tail_SS(process.priority) ← process_SS;
x ← atomic_read(port.lock);
atomic_write(port.lock) ← x xor 1;
perform process suspension action;
# IP continues to point at receive inst
x ← atomic_read(current_process.lock);
atomic_write(current_process.lock) ← x xor 1;
perform dispatch action;
else if port is fifo
    then dequeue first message;
    else dequeue first message from highest-priority
        nonempty queue;
dst ← message_SS;
x ← atomic_read(port.lock);
atomic_write(port.lock) ← x xor 1;
endif;
```

Faults: STANDARD

Example: receive g8, g3 # receives message from port g8
# and store message in g3

Opcode: receive 656 REG

See Also: conrec, send





**remr, remrl**

**Mnemonic:** **remr** Remainder Real  
**remrl** Remainder Long Real

**Format:** **remr\*** *src1*, *src2*, *dst*  
 freg/flit freg/flit freg

**Description:** Divides *src2* by *src1* and stores the remainder in *dst*. The sign of the result (if nonzero) is the same as the sign of *src2*.

For the **remrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						Src2
		$-\infty$	-F	-0	+0	+F	$+\infty$	
		$-\infty$	*	*	*	*	*	NaN
		-F	src2	-F or -0	**	**	-F or -0	src2
		-0	-0	-0	*	*	-0	NaN
Src2	+0	+0	+0	*	*	+0	+0	NaN
	+F	src2	+F or +0	**	**	+F or +0	src2	NaN
	$+\infty$	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

- F Means finite-real number.
- \* Indicates floating invalid-operation exception.
- \*\* Indicates floating zero-divide exception.

When the result is 0, its sign is the same as that of *src2*. When the *src1* is  $\infty$ , the result is equal to the *src2*.

The result of this operation is always exact if the destination format is at least as wide as the *src2* and *src1*.

**remr, remrl**

The remainder provided with the **remr** and **remrl** instructions is different from the remainder described in the IEEE floating-point standard. The difference is related to how the quotient (N) of the expression (*src2/src1*) is determined.

As shown below in the action statement, N for the **remr** and **remrl** instructions is the nearest integer value obtained when the exact result (E) of the expression (*src2/src1*) is truncated toward zero. N will always be less than or equal to the absolute value of E.

For the IEEE standard, N is simply the nearest integer value to E. Here, N may be less than, equal to, or greater than the absolute value of E.

To help determine the IEEE remainder from the result given by the **remr** and **remrl** instructions, the following information about the quotient is given in the arithmetic-status field in the arithmetic controls:

Arithmetic Status Bit	Meaning
6	Q1, the next-to-last quotient bit
5	Q0, the last quotient bit
4	QR, the value the next quotient bit would have if one more reduction were performed (the "round" bit of the quotient)
3	QS, set if the remainder after the QR reduction would be nonzero (the "sticky" bit of the quotient)

The information can then be used to determine the IEEE standard remainder, as shown in the example on the next page.

**Action:**

```
dst ← src2 - (N * src1);  
# where N = truncate (src2/src1).  
# Here, (src2/src1) is truncated  
# toward zero to the nearest integer.
```

## remr, remrl

**Faults:** STANDARD Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow Result is too large for destination format.

Floating Underflow Result is too small for destination format.

Floating Zero Divide The *src1* operand is 0.

Floating Invalid Operation The *src2* operand is  $\infty$ .

One or more operands are an SNaN value.

Floating Inexact Result cannot be represented exactly in destination format.

**Example:**

```
# z = ieee_rem(x, y)
# z is in g0,g1; x is in g0,g1; y is in g2,g3
_ieee_rem:
    remrl g2, g0, g0
    modac 0, 0, g4
    bbc 4, g4, 2f
    # QR=0, implies g0 < y/2 and z=g0
    bbs 3, g4, 1f
    # QR=1, QS=1, implies g0 > y/2 and z=g0-y
    bbc 5, g4, 2f
    # QR=1, QS=0, Q0=0, implies g0=y/2 and z=g0
1: clrbit 31, g3, g2 # |y|
   subrl g2, g0, g0
2: ret
```

**Opcode:** **remr** 683 REG  
**remrl** 693 REG

**See Also:** remi, modi

## resumprcs

**Mnemonic:** resumprcs Resume Process

**Format:** resumprcs *src*  
reg  
SS

**Description:** Switches the processor from one process to another process. The SS of the new process is specified with the *src* operand.

The processor must be in the supervisor mode to execute this instruction.

Any state information for the current process that has been cached on the processor chip, such as the PCB and the stack frames, is discarded (i.e., not updated in memory, not unlocked). Thus, to save the state of the current process, the **resumprcs** instruction should be preceded by a **saveprcs** instruction.

The **saveprcs** and **resumprcs** instructions are similar to the **save** and **resume** functions in most UNIX kernels. These instructions allow task (or process) switching without using the processor's automatic dispatching mechanism.

**Action:** if *src1* is not a SS to a PCB  
then raise Type Mismatch Fault;  
endif;  
perform process-bind action

**Faults:** STANDARD, Type Mismatch

**Example:** resumprcs r4 # processor is bound  
# to process  
# specified in r4

**Opcode:** resumprcs 664 REG

**See Also:** saveprcs

**ret****Mnemonic:**    **ret**                    Return**Format:**        **ret**

**Description:** Returns process control to the calling procedure. The current stack frame (i.e., that of the called procedure) is deallocated and the FP is changed to point to the stack frame of the calling procedure. Instruction execution is continued at the instruction pointed to by the RIP in the calling procedure's stack frame, which is the instruction immediately following the call instruction.

As shown in the action statement below, the return status field and prereturn trace flag determine the action that the processor takes on the return. These fields are contained in bits 0 through 3 of register\_r0 of the calling procedure's local registers.

Refer to Chapter 4 for further discussion of the **ret** instruction.

**Action:** wait for any uncompleted instructions to finish;  
           **case return\_status is**

2#000#: FP ← PFP;  
           free current register\_set;  
           **if** register\_set (FP) not allocated  
               **then** retrieve from memory(FP);  
           **end if**;  
           IP ← RIP;

2#001#: x ← memory(FP-16);  
           y ← memory(FP-12);  
           **go to** case 000 action;  
           arithmetic\_controls ← y;  
           **if** execution\_mode = supervisor  
               **then** process\_controls ← x;  
           **end if**;

2#010#: **if** execution\_mode ≠ supervisor  
           **then go to** case 000 action;  
           **else** process\_controls.T ← 0;  
               execution\_mode ← user;  
               **go to** case 000 action;  
           **end if**;

# ret

	Mnemonic: Rotate
	Format: Rotate
	Description: Copies src to dst and rotates the result by the specified amount. (The bit shifted off the left end of the word is inserted at the right end of the word. The left operand can range from 0 to 31.)
	Action: $dst \leftarrow rotate(dst, mod \ 32)$
	Faults: STANDARD
	Example: Rotate r4, r8, #16; Rotate r4, r8, #16; Rotate r4, r8, #16;
	Opcode: Rotate
	See Also: SHIFT

**Faults:** STANDARD

**Example:** ret # process control returns to  
# calling procedure  
# environment

**Opcode:** ret 0A CTRL

**See Also:** call, calls, callx



## rotate

**Mnemonic:** rotate Rotate

**Format:** rotate *len*, *src*, *dst*  
reg/lit reg/lit reg

**Description:** Copies *src* to *dst* and rotates the bits in the resulting *dst* operand to the left (toward higher significance). (The bits shifted off the left end of the word are inserted at the right end of the word.) The *len* operand specifies the number of bits that the *dst* operand is rotated. The *len* operand can range from 0 to 31.

This instruction can also be used to rotate bits to the right. Here, the number of bits the word is to be rotated right is subtracted from 32 to get the *len* operand.

**Action:**  $dst \leftarrow \text{rotate}(len \bmod 32, src)$

**Faults:** STANDARD

**Example:** rotate r4, r8, r12 # r12 ← r8  
# with bits rotated  
# r4 bits to left

**Opcode:** rotate 59D REG

**See Also:** SHIFT

## roundr, roundrl

**Mnemonic:** **roundr** Round Real  
**roundrl** Round Long Real

**Format:** **roundr\*** *src*, *dst*  
*freg/flit* *freg*

**Description:** Rounds *src* to the nearest integral value, depending on the rounding mode, and stores the result in *dst*.

For the **roundrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

If the *src* operand is  $\infty$  the result is *src*. If the *src* operand is not an integral value, a floating-inexact exception is raised.

**Action:**  $dst \leftarrow \text{round\_to\_integral\_value}(src);$

**Faults:** **STANDARD:** Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

**Example:** `roundrl r4, r10`  
`# r10, r11 ← r4, r5 rounded`

**Opcode:** **roundr** 68B REG  
**roundrl** 69B REG

# saveprcs

**Mnemonic:** saveprcs Save Process

**Format:** saveprcs

**Description:** Updates the state of the current process in memory by saving that part of the process state that is cached on the processor chip during the execution of the process. The part of the process state that is cached includes part of the PCB and any cached local-register frames. The process is not unlocked and continues to execute with its cached state.

The processor must be in the supervisor mode to execute this instruction.

The **saveprcs** and **resumprcs** instructions are similar to the **save** and **resume** functions in most UNIX kernels. These instructions allow task (or process) switching without using the processor's automatic dispatching mechanism.

The primary function of the **saveprcs** instruction is to save the state of a process prior to switching processes using the **resumprcs** instruction.

**Action:** if PRCB.processor\_controls.state = process\_executing  
then perform process-suspension action  
else flush any local register sets;  
endif;

**Faults:** STANDARD

**Opcode:** saveprcs 666 REG

**See Also:** resumprcs

**Example:**

# r10, r11 → r4, r5 rounded  
roundr1 r4, r10

**Opcode:**

roundr ~ 68B  
roundr1 69B  
REG  
REG

scaler, scalerl

Mnemonic: scaler Scale Real  
scalerl Scale Long Real

Format: scaler\* src1, src2, dst  
reg/lit freg/flit freg

Description: Multiplies *src2* by 2 to the power of *src1* and stores the result in *dst*. The *src1* operand is an integer; whereas, *src2* and *dst* are reals.

For the **scalerl** instruction, if the *src2* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1			
Src2		-N	0	+ N	
	-∞	-∞	-∞	-∞	
	-F	-F	-F	-F	
	-0	-0	-0	-0	
	+ 0	+ 0	+ 0	+ 0	
	+ F	+ F	+ F	+ F	
	+ ∞	+ ∞	+ ∞	+ ∞	
	NaN	NaN	NaN	NaN	

Notes:  
F Means finite-real number.  
N Means integer.

In most cases, only the exponent is changed and the mantissa (fraction) remains unchanged. However, when the *src2* operand is a denormalized value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

## scaler, scalerl

Refer to the sections titled "Floating Overflow Exception" and "Floating Underflow Exception" in Chapter 7 for further discussion of how overflow and underflow are handled.

**Action:**  $dst \leftarrow src2 * (2^{src1})$

**Faults:** STANDARD Refer to the discussion of faults at the beginning of this chapter.

**Floating Reserved Encoding** One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Zero Divide	The <i>src1</i> operand is 0.
Floating Invalid Operation	One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

**Example:**

```
scalerl g6, g2, fp0
# fp0 ← g2, g3 * 2^g6
```

**Opcode:**

scaler	677	REG
scalerl	676	REG

**See Also:** mulr

scanbit

Mnemonic:	scanbit	Scan For Bit			
Format:	scanbit	src, reg/lit	dst, reg		
Description:	Searches the <i>src</i> value for the most-significant set bit (1 bit). If a most-significant 1 bit is found, its bit number is stored in <i>dst</i> and the condition code is set to 010 <sub>2</sub> . If the <i>src</i> value is zero, all 1's are stored in <i>dst</i> and the condition code is set to 000 <sub>2</sub> .				
Action:	<pre> dst ← 16#FFFFFFFF#; AC.cc ← 2#000#; for i in 31..0 reverse loop     if (src and 2^i) ≠ 0     then         dst ← i;         AC.cc ← 2#010#;         exit;     end if; end loop; </pre>				
Faults:	STANDARD				
Example:	<pre> # assume g8 is nonzero scanbit g8, g10 # g10 ← bit number of # most-significant set bit # in g8; AC.cc ← 010 </pre>				
Opcode:	scanbit	641	REG		
See Also:	spanbit				



scanbyte

Mnemonic: scanbyte Scan Byte Equal

Format: scanbyte src1, src2  
reg/lit reg/lit

Description: Performs a byte-by-byte comparison of *src1* and *src2* and sets the condition code to 010<sub>2</sub> if any two corresponding bytes are equal. If no corresponding bytes are equal, the condition code is set to 000<sub>2</sub>.

Action: if (*src1* and 16#000000FF#) = (*src2* and 16#000000FF#) or  
(*src1* and 16#0000FF00#) = (*src2* and 16#0000FF00#) or  
(*src1* and 16#00FF0000#) = (*src2* and 16#00FF0000#) or  
(*src1* and 16#FF000000#) = (*src2* and 16#FF000000#)  
then AC.cc ← 2#010#;  
else AC.cc ← 2#000#;  
endif;

Faults: STANDARD

Example: # assume r9 = 0x11AB1100  
scanbyte 0x00AB0011, r9  
# AC.cc ← 010

Opcode: scanbyte 5AC REG

See Also: cmpstr, fill, movqstr, movstr

**schedprcs**

Mnemonic:	schedprcs	Schedule Process			
Format:	schedprcs	src	reg	SS	

**Description:** Sends a process to its dispatching port. The *src* operand specifies the SS of the PCB for the process to be scheduled. If the preempt bit in PCB of the process is set and if its priority is higher than the currently running process, a preemption action is initiated. Otherwise, the process is enqueued at the head of its priority queue at the dispatching port.

The processor must be in the supervisor mode to execute this instruction. The SS of the dispatching port and the priority of the process are determined from the process's PCB; then rescheduled at its dispatching port, the message is enqueued at the end of the queue of the priority specified in the *src* operand. The *src* operand can range from 0 to 31.

**Action:** perform unblock action on process specified with *src*;

**Faults:** STANDARD

**Example:** schedprcs g3  
# process specified in g3 is scheduled

**Opcode:** schedprcs 665 REG

**See Also:** sendserv

**Action:**

```
atomic_write(port.lock) ← x xor 1;
x ← atomic_read(port.lock);
port.queue_tail_SS(src) ← src;
else enqueue src on port.queue_tail_SS(src mod 32);
then if port is fifo
  if port.Q = 0
    else atomic_write(port.lock) ← x or 1;
  go to send;
then atomic_write(port.lock) ← x;
if least_significant_bit(x) = 1
  x ← atomic_read(port.lock);
```

**send**

<b>Mnemonic:</b>	<b>send</b>	<b>Send</b>			
<b>Format:</b>	<b>send</b>	<i>dst</i> , reg SS	<i>src1</i> , reg/lit SS	<i>src2</i> reg SS	

**Description:** Sends a message to a communications port. The *src2* operand specifies the SS of the message being sent and the *dst* operand specifies the SS of the port the message is to be sent to.

The processor must be in the supervisor mode to execute this instruction.

If the port is a priority-type port, the message is handled as follows. If there are processes enqueued at the port, the message is bound to the process at the head of the highest priority queue that has queued processes. The process is then rescheduled at its dispatching port. If there are no processes enqueued at the port, the message is enqueued at the end of the queue of the priority specified in the *src1* operand. The *src1* operand can range from 0 to 31.

If the port is a FIFO port, the message is handled in the same way, except that the priority operand (*src1*) is ignored.

The message is bound to a process by writing the SS of the message in the receive message field of the process's PCB.

When the process is rescheduled, a preemption action is initiated if the preempt bit in the process's PCB is set and if the process has a higher priority than the currently running process.

**Action:**

```

x ← atomic_read(port.lock);
if least_significant_bit(x) = 1
  then atomic_write(port.lock) ← x;
  go to send;
else atomic_write(port.lock) ← x or 1;
  if port.Q = 0
    then if port is fifo
      then enqueue src2 on port
        port.queue_tail_SS ← src2;
      else enqueue src2 on port.queue(src1 mod 32);
        port.queue_tail_SS(src1 mod 32) ← src2;
    x ← atomic_read(port.lock);
    atomic_write(port.lock) ← x xor 1;

```

send

Mnemonic: send

Format:  $\text{send } \text{src}$

Description:  $\text{send } \text{src}$   
Sends the current  $\text{src}$  to the port specified in  $\text{src}$ . If the port is empty, the message is sent immediately. If the port is not empty, the message is queued and the processor is blocked until the port is empty. The processor must be in the supervisor mode to execute this instruction.

**else if** port is fifo  
    **then** dequeue first process;  
    **else** dequeue first process from highest-priority nonempty queue;  
    dequeued\_process.received\_message  $\leftarrow \text{src2}$ ;  
     $x \leftarrow \text{atomic\_read}(\text{port.lock})$ ;  
     $\text{atomic\_write}(\text{port.lock}) \leftarrow x \text{ xor } 1$ ;  
    perform unblock action on dequeued process;  
**end if**;

**Faults:** STANDARD

**Example:**  $\text{send } \text{g8}, 21, \text{g2}$   
# message with the SS given  
# in g2 is sent to the priority  
# port with the SS given in g8;  
# if the port is empty, the  
# message is queued at  
# priority queue 21

**Opcode:** send 662 REG

**See Also:** condrec, receive

## sendserv

**Mnemonic:**    **sendserv**    Send Service

**Format:**        **sendserv**    *src*  
    *reg*  
    **SS**

**Description:**    Suspends the current process and sends the SS of its PCB as a message to the port specified in *src*. If the port is a FIFO port, the process SS is queued at the end of the queue.

The processor must be in the supervisor mode to execute this instruction.

If the port is a priority port, the process SS is queued at the end of the queue for its specified priority or given to the highest priority process waiting at the priority port, if one is available. The priority of the process is determined from the Process Controls word in the PCB for the process.

**Action:**

```
perform process suspension action;
x ← atomic_read(port.lock);
if least_significant_bit(x) = 1
  then atomic_write(port.lock) ← x;
  go to sendserv;
else atomic_write(port.lock) ← (x or 1);
if port.Q = 0
  then if port is fifo
    then enqueue current_process as message on port
         port.queue_tail_SS ← current_process_SS;
    else enqueue current_process as message on
         port.queue(current_process.priority);
         port.queue_tail_SS(current_process.priority) ←
         current_process_SS;
  x ← atomic_read(port.lock);
  atomic_write(port.lock) ← x xor 1;
  x ← atomic_read(current_process.lock);
  perform dispatch action;
```

sendserv

else if port is fifo  
  then dequeue first process;  
  else dequeue first process from highest-priority  
      nonempty queue;  
      dequeued\_process.received\_message ← current\_process.SS;  
      x ← atomic\_read(port.lock);  
      atomic\_write(port.lock) ← x xor 1;  
      x ← atomic\_read(current\_process.lock);  
      atomic\_write(current\_process.lock) ← x xor 1;  
      perform steps 1..3 of unblock action on dequeued process;  
      perform dispatch action;  
end if;

Faults: STANDARD

Example: sendserv r4  
          # process is suspended and sent  
          # to the port with the SS  
          # given in r4

Opcode:       sendserv   663       REG

See Also:     schedprcs



# setbit

**Mnemonic:**    **setbit**           Set Bit

**Format:**       **setbit**       *bitpos*,    *src*,    *dst*  
                                  reg/lit    reg/lit    reg

**Description:**   Copies the *src* value to *dst* with one bit set. The *bitpos* operand specifies the bit to be set.

**Action:**        *dst* ← *src* or  $2^{(bitpos \bmod 32)}$ ;

**Faults:**        STANDARD

**Example:**       setbit 15, r9, r1  
                  # r1 ← r9 with bit 15 set

**Opcode:**       **setbit**       583       REG

**See Also:**       alterbit, chkbit, clrbit, notbit

## SHIFT

<b>Mnemonic:</b>	<b>shlo</b>	Shift Left Ordinal
	<b>shro</b>	Shift Right Ordinal
	<b>shli</b>	Shift Left Integer
	<b>shri</b>	Shift Right Integer
	<b>shrdi</b>	Shift Right Dividing Integer

<b>Format:</b>	<b>sh*</b>	<i>len</i> ,	<i>src</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg

**Description:** Shifts *src* left or right by the number of bits indicated with the *len* operand and stores the result in *dst*. Bits shifted beyond the register boundary are discarded. For values of **len** greater than 32, the processor interprets the value as 32.

The **shlo** instruction shift zeros in from the least-significant bit, and the **shro** instruction shifts zeros in from the most-significant bit. These instructions are equivalent to **mulo** and **divo** by the power of 2, respectively.

The **shli** instruction shifts zeros in from the least-significant bit; if the bits shifted out are not the same as the sign bit, an overflow fault is generated. If overflow occurs, the sign of the result is the same as the sign of the *src* operand.

The **shri** instruction performs a conventional arithmetic shift-right operation by shifting the sign bit in from the most-significant bit. When this instruction is used to divide an negative integer operand by the power of 2, it produces an incorrect quotient. (The discarding of the bits shifted out has the effect of rounding the result toward negative.)

The **shrdi** instruction is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result if the bits shifted out are non-zero and the operand is negative, which produces the correct result for negative operands.

The **shli** and **shrdi** instructions are equivalent to **muli** and **divi** by the power of 2.

# SHIFT

Action:	shlo:	if $len < 32$ then $dst \leftarrow src * 2^{len}$ ; else $dst \leftarrow 0$ ; end if;	Mnemonic:
	shro:	if $len < 32$ then $dst \leftarrow src / 2^{len}$ ; else $dst \leftarrow 0$ ; end if;	
	shli:	$dst \leftarrow src * 2^{len}$ ;	Description:
	shri:	if $src \geq 0$ then if $len < 32$ then $dst \leftarrow src / 2^{len}$ ; else $dst \leftarrow 0$ ; else if $len < 32$ then $dst \leftarrow (src - 2^{len} + 1) / 2^{len}$ ; else $dst \leftarrow -1$ ; end if; end if;	
	shrdi:	$dst \leftarrow src / 2^{len}$ ;	
Faults:	STANDARD, Integer Overflow		
Example:	shli 13, g4, r6 # g6 $\leftarrow$ g4 shifted left 13 bits		
Opcode:	shlo	59C	REG
	shro	598	REG
	shli	59E	REG
	shri	59B	REG
	shrdi	59A	REG

See Also: divi, muli, rotate

## signal

**Mnemonic:** signal Signal

**Format:** signal *dst*  
reg  
SS

**Description:** Unblocks (dequeues) a process from the semaphore queue, if there are processes enqueued. If there is no process queued at the semaphore, the semaphore count is incremented by one. The *dst* operand gives the SS of the semaphore being signaled. If a process is dequeued, it is rescheduled at its dispatching port. The processor must be in the supervisor mode to execute this instruction.

**Action:**

```

x ← atomic_read (semaphore.lock);
if least_significant_bit(x) = 1
then atomic_write (semaphore.lock) ← x;
    go to signal;
else atomic_write (semaphore.lock) ← x or 1;
    if semaphore.tail ≠ 0
    then dequeue first process;
        x ← atomic_read (semaphore.lock);
        atomic_write (semaphore.lock) ← x xor 1;
        perform unblock action
            on dequeued process;
    else semaphore.count ←
        semaphore.count + 1;
        x ← atomic_read (semaphore.lock);
        atomic_write (semaphore.lock) ← x xor 1;
end if;

```

**Faults:** STANDARD

**Example:** signal r8  
# semaphore with SS given  
# in r8 is signaled

**Opcode:** signal 66A REG

**See Also:** condwait, wait

**sinr, sinrl**

**Mnemonics:** **sinr** Sine Real  
**sinrl** Sine Long Real

**Format:** **sinr\*** *src*, *dst*  
 freg/flit freg

**Description:** Calculates the sine of *src* and stores the result in *dst*. The *src* value is an angle given in radians. The resulting *dst* value is in the range -1 to +1, inclusive.

For the **sinrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the sine of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src	Dst
$-\infty$	*
-F	-1 to +1
-0	-0
+0	+0
+F	-1 to +1
$+\infty$	*
NaN	NaN

Notes:

- F Means finite-real number
- \* Indicates floating invalid-operation exception

In the trigonometric instructions, the 80960MC uses a value for  $\pi$  with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 7 titled "Pi" gives this  $\pi$  value, along with some suggestions for representing this value in a program.

**Action:**  $dst \leftarrow \text{sine}(src);$

sinr, sinrl

Faults:	STANDARD	Refer to the discussion of faults at the beginning of this chapter.
	Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.
	The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.	
	Floating Invalid Operation	The <i>src</i> operand is $\infty$ .
		One or more operands is an SNaN value.
	Floating Inexact	Result cannot be represented exactly in destination format.
Example:	<pre>sinrl g6, g0 # sine of value in g6, g7 # is stored in g0, g1</pre>	
Opcode:	sinr      68C      REG	
	sinrl     69C      REG	
See Also:	cosr, tanr	



## spanbit

**Mnemonic:** spanbit Span Over Bit

**Format:** spanbit src, dst  
reg/lit reg

**Description:** Searches the *src* value for the most-significant clear bit (0 bit). If a most-significant 0 bit is found, its bit number is stored in *dst* and the condition code is set to 010<sub>2</sub>. If the *src* value is all 1's, all 1's are stored in *dst* and the condition code is set to 000<sub>2</sub>.

**Action:**

```
dst ← 16#FFFFFFFF#;
AC.cc ← 2#000#;
for i in 31..0 reverse loop
  if (src and 2^i) = 0
  then
    dst ← i;
    AC.cc ← 2#010#;
    exit;
  end if;
end loop;
```

**Faults:** STANDARD

**Example:**

```
# assume r2 is not 0xffffffff
spanbit r2 r9
# r9 ← bit number of
# most-significant clear bit
# in r2; AC.cc ← 010
```

**Opcode:** spanbit 640 REG

**See Also:** scanbit

sqrtr, sqrtrl

Mnemonic:   sqrtr       Square Root Real  
              sqrtrl     Square Root Long Real

Format:     sqrtr\*     src,     dst  
              freg/flit   freg

Description:   Calculates the square root of *src* and stores it in *dst*.  
For the **sqrtrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).  
The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src	Dst
-∞	*
-F	*
-0	-0
+ 0	+ 0
+ F	+ F
+ ∞	+ ∞
NaN	NaN

- Notes:
- F   Means finite-real number
  - \*   Indicates floating invalid-operation exception

With these instructions, it is not possible to raise a floating overflow or floating underflow fault unless the *src* operand is in a floating-point register and the *dst* operand is not.

Action:       dst ← sqrt (src);



## STORE

<b>Mnemonic:</b>	<b>st</b>	Store	80	st	Opcode
	<b>stob</b>	Store Ordinal Byte	82	stob	
	<b>stos</b>	Store Ordinal Short	8A	stos	
	<b>stib</b>	Store Integer Byte	C2	stib	
	<b>stis</b>	Store Integer Short	CA	stis	
	<b>stl</b>	Store Long	9A	stl	
	<b>stt</b>	Store Triple	A2	stt	
	<b>stq</b>	Store Quad	B2	stq	

<b>Format:</b>	<b>src*</b>	<i>src</i> ,	<i>dst</i>	LOAD, MOVE	See Also:
		reg	mem		

**Description:** Copies a byte or string of bytes from a register or group of registers to memory. The *src* operand specifies a register or the first (lowest numbered) register of successive registers.

The *dst* operand specifies the address of the memory location where the byte or the first byte of a string of bytes is to be stored. The full range of addressing modes may be used in specifying *dst*. (Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.)

The **stob** and **stib**, and **stos** and **stis** instructions store a byte and half word, respectively, from the low order bytes of the *src* register. The **st**, **stl**, **stt**, and **stq** instructions copy 4, 8, 12, and 16 bytes, respectively, from successive registers to memory.

For the **stl** instruction, *src* must specify an even numbered register (e.g., g0, g2, ..., g12). For the **stt** and **stq** instructions, *src* must specify a register number that is a multiple of four (e.g., g0, g4, g8).

**Action:** memory (*dst*)  $\leftarrow$  *src*;

**Faults:** STANDARD, Integer Overflow Fault (**stib** and **stis** instructions only)

**Example:**

```

st g2, 1256 (g6)
# word beginning at offset
# 1256 + (g6)  $\leftarrow$  g2

```

# STORE

Opcode:	st	92	MEM	st	Mnemonic:
	stob	82	MEM	stop	
	stos	8A	MEM	stos	
	stib	C2	MEM	stib	
	stis	CA	MEM	stis	
	stl	9A	MEM	stl	
	stt	A2	MEM	stt	
	stq	B2	MEM	stp	

See Also: LOAD, MOVE

**Description:** Copies a byte or string of bytes from a register or group of registers to memory. The src operand specifies a register or the first (lowest numbered) register of successive registers.

The dst operand specifies the address of the memory location where the byte or the first byte of a string of bytes is to be stored. The full range of addressing modes may be used in specifying dst. (Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.)

The stop and stib, and stos and stis instructions store a byte and half word, respectively, from the low order bytes of the src register. The stl, stt, and stp instructions copy 4, 8, 12, and 16 bytes, respectively, from successive registers to memory.

For the stl instruction, src must specify an even numbered register (e.g., %0, %2, ..., %12). For the stt and stp instructions, src must specify a register number that is a multiple of four (e.g., %0, %4, %8).

**Action:** memory (dst) ← src;

**Faults:** STANDARD, Integer Overflow Fault (stib and stis instructions only)

**Example:** # 1256 + (%0) → %2  
# word beginning at offset  
at %2, 1256 (%0)

# subc

Mnemonic:	subc	Subtract Ordinal With Carry		
Format:	subc	src1, reg/lit	src2, reg/lit	dst reg
Description:	<p>Subtracts (<i>src1</i> - 1) from <i>src2</i>, adds bit 1 of the condition code (used here as a carry bit), and stores the result in <i>dst</i>. If the ordinal subtraction results in a carry, bit 1 of the condition code is set.</p> <p>This instruction can also be used for integer subtraction. Here, if integer subtraction results in an overflow, bit 0 of the condition code is set.</p> <p>The <b>subc</b> instruction does not distinguish between ordinals and integers: it sets bits 0 and 1 of the condition code regardless of the data type.</p>			
Action:	<p># Let the value of the condition code be xCx, <math>dst \leftarrow src2 - (src1 - 1) + C;</math> <math>AC.cc \leftarrow 2\#0CV\#;</math> # C is carry from ordinal subtraction. # V is 1 if integer subtraction would have generated # an overflow.</p>			
Faults:	STANDARD			
Example:	<pre>subc g5, g6, g7 # g7 ← g6 - (g5 - 1) # + Carry Bit</pre>			
Opcode:	subc	5B2	REG	
See Also:	addc			



# subi, subo

<b>Mnemonic:</b>	<b>subi</b> <b>subo</b>	Subtract Integer Subtract Ordinal
<b>Format:</b>	<b>sub*</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit reg/lit reg
<b>Description:</b>	Subtracts <i>src1</i> from <i>src2</i> and stores the result in <i>dst</i> . The binary results from these two instructions are identical. The only difference is that <b>subi</b> can signal an integer overflow.	
<b>Action:</b>	$dst \leftarrow src2 - src1$ ;	
<b>Faults:</b>	STANDARD, Integer Overflow ( <b>subi</b> instruction only)	
<b>Example:</b>	subi g6, g9, g12 # g12 $\leftarrow$ g9 - g6	
<b>Opcode:</b>	<b>subi</b> 593 <b>subo</b> 592	REG REG
<b>See Also:</b>	addi, addr, subc, subr	

**Mnemonic:**     **subr**           Subtract Real  
                   **subrl**          Subtract Long Real

**Format:**       **subr\***     *src1*,     *src2*,     *dst*  
                                   freg/flit   freg/flit   freg

**Description:**   Subtracts *src1* from *src2* and stores the result in *dst*.

For the **subrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when subtracting various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						
Src2		$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
	$-\infty$	*	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN
	-F	$+\infty$	$\pm F$ or $\pm 0$	src2	src2	-F	$-\infty$	NaN
	-0	$+\infty$	src1	$\pm 0$	-0	src1	$-\infty$	NaN
	+0	$+\infty$	src1	+0	$\pm 0$	src1	$-\infty$	NaN
	+F	$+\infty$	+F	src2	src2	$\pm F$ or $\pm 0$	$-\infty$	NaN
	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

- F Means finite-real number.
- \* Indicates floating invalid-operation exception.

When the difference between two operands of like sign is zero, the result is +0, except for the round toward  $-\infty$  mode, in which case the result is -0. This instruction also guarantees that +0 - (-0) = +0, and that -0 - (+0) = -0.

When one source operand is  $\infty$ , the result is  $\infty$  of the expected sign. If both source operands are  $\infty$  of the same sign, an invalid-operation exception is raised.

# subr, subrl

**Action:**  $dst \leftarrow src2 - src1;$

**Faults:** STANDARD

Floating Reserved Encoding

Refer to the discussion of faults at the beginning of this chapter.

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow

Result is too large for destination format.

Floating Underflow

Result is too small for destination format.

Floating Invalid Operation

Source operands are infinities of like sign.

One or more operands are an SNaN value.

Floating Inexact

Result cannot be represented exactly in destination format.

**Example:** `subrl g6, fp0, fp1`  
`# fp1  $\leftarrow$  fp0 - g6, g7`

**Opcode:** `subr` 78D REG  
`subrl` 79D REG

**See Also:** `subi, subc, addr`

**syncf**

**Mnemonic:** syncf Synchronize Faults

**Format:** syncf

**Description:** Waits for any faults to be generated associated with any prior uncompleted instructions.

**Action:** if arithmetic\_controls.nif  
then;  
else wait until no imprecise faults can occur  
associated with any uncompleted instructions;

end if;

**Faults:** STANDARD

**Example:** ld xyz, g6  
addi r6, r8, r8  
syncf  
and g6, 0xFFFF, g8  
# the syncf instruction insures that any faults  
# that may occur during the execution of the  
# ld and addi instructions occur before the  
# and instruction is executed

**Opcode:** syncf 66F REG

**See Also:** mark, fmark

**synld****Mnemonic:** **synld** Synchronous Load**Format:** **synld** *src*, *dst*  
reg reg  
addr**Description:** Copies a word from the memory location specified with *src* into *dst* and waits for the completion of all memory operations, including those initiated prior to the **synld** instruction. When the load has been successfully completed, the condition code is set to 010<sub>2</sub>.

The primary function of this instruction is for reading IAC messages, the IAC Message Control word, or the IAC Interrupt Control Register. However, this instruction is not restricted to IAC applications. It may be used when it is important to guarantee the completion of the load operation before proceeding or to avoid a bad-access fault.

The setting of the condition code indicates whether or not the load was completed successfully. If the load operation results in a bad access condition (e.g., reading an AP-bus interconnect register), the condition code is set to 000<sub>2</sub>, but the bad-access fault is not raised.

**Action:**

```

if PRCB.addressing_mode = physical
  then tempa ← src;
  else tempa ← physical_address(src);
end if;
tempa ← tempa and 16#FFFFFFFC#; # force alignment
if tempa = 16#FF000004#
  then dst ← interrupt_control_reg;
  AC.cc ← 2#010#;
else dst ← memory(tempa);
  if bad_access
    then AC.cc ← 2#000#;
    else AC.cc ← 2#010#;
  end if;
end if;

```

**Faults:** STANDARD

synld

**Example:**     lda 0xff000004, g8  
                   # g8 ← address of interrupt-control register  
                   synld g8, g9  
                   # g9 ← contents of interrupt-control register  
                   # AC.cc = 010

**Opcode:**     synld     615     REG

**See Also:**     synmov

**Description:** Copies 1 (synmov), 2 (synmov), or 4 (synmov) copies of the data specified by the register or memory location specified with r/c to the memory location specified with d/a and waits for the completion of all memory operations, including those initiated prior to this instruction. When the move has been successfully completed, the condition code is set to 010.

The r/c and d/a operands specify the address of the first (lowest address) word. These addresses should be for word boundaries (synmov), double-word boundaries (synmov), or quad-word boundaries (synmov). If not, the processor forces alignment to these boundaries.

The primary function of these instructions is for sending IAC messages. However, this instruction is not restricted to IAC applications. It may be used when it is important to guarantee the completion of the move operation before proceeding or to avoid a Bad Access Fault.

The setting of the condition code indicates whether or not the move was completed successfully. If the move operation results in a bad access condition (e.g., sending an IAC message to a non-existent agent on the AP-bus), the condition code is set to 000, but the Bad Access Fault is not raised.

Address FF000010<sub>16</sub> is used to send an IAC message to the processor upon which the instruction is executed. Refer to Chapter 11 for further information about sending internal IAC messages.

**Action:**     synmov

If PRCB.addressing\_mode = physical  
                   then tempa ← d/a;  
                   # d/a is used as a physical address



## synmov, synmovl, synmovq

**Mnemonic:**    **synmov**    Synchronous Move  
                  **synmovl**    Synchronous Move Long  
                  **synmovq**    Synchronous Move Quad

**Format:**        **synmov\***    *dst*,        *src*  
                                      reg        reg  
                                      addr       addr

**Description:**    Copies 1 (**synmov**), 2 (**synmovl**), or 4 (**synmovq**) words from the memory location specified with *src* to the memory location specified with *dst* and waits for the completion of all memory operations, including those initiated prior to this instruction. When the move has been successfully completed, the condition code is set to 010<sub>2</sub>.

The *src* and *dst* operands specify the address of the first (lowest address) word. These addresses should be for word boundaries (**synmov**), double-word boundaries (**synmovl**), or quad-word boundaries (**synmovq**). If not, the processor forces alignment to these boundaries.

The primary function of these instructions is for sending IAC messages. However, this instruction is not restricted to IAC applications. It may be used when it is important to guarantee the completion of the move operation before proceeding or to avoid a Bad Access Fault.

The setting of the condition code indicates whether or not the move was completed successfully. If the move operation results in a bad access condition (e.g., sending an IAC message to a non-existent agent on the AP-bus), the condition code is set to 000<sub>2</sub>, but the Bad Access Fault is not raised.

Address FF000010<sub>16</sub> is used to send an IAC message to the processor upon which the instruction is executed. Refer to Chapter 11 for further information about sending internal IAC messages.

**Action:**            **synmov:**

```
if PRCB.addressing_mode = physical
  then tempa ← dst;
  # dst is used as a physical address
```

## synmov, synmovl, synmovq

```

else tempa ← physical_address (dst);
# dst translated into a physical address
end if;
tempa ← tempa and 16#FFFFFFFC#;
# force alignment
if tempa = 16#FF000004#
then interrupt_control_reg ← memory (src)
AC.cc ← 2#010#;
else temp ← memory (src);
memory (tempa) ← temp;
# write operations into memory (tempa) are
# interpreted as noncacheable
wait for completion;
if bad_access
then AC.cc ← 2#000#;
else AC.cc ← 2#010#;
end if;
end if;

synmovl:
if PRCB.addressing_mode = physical
then tempa ← dst;
# dst is used as a physical address
else tempa ← physical_address (dst);
# dst is translated into as a physical address
end if;
tempa ← tempa and 16#FFFFFFF8#; # force alignment
temp ← memory (src);
memory (tempa) ← temp;
# write operations into memory (tempa) are interpreted
# as noncacheable
wait for completion;
if bad_access
then AC.cc ← 2#000#;
else AC.cc ← 2#010#;
end if;

```

## synmov, synmovl, synmovq

```

synmovq:
    if PRCB.addressing_mode = physical
    then tempa ← dst;
    # dst is used as a physical address
    else tempa ← physical_address (dst);
    # dst is translated into as a physical address
    end if;
    tempa ← tempa and 16#FFFFFFF0#; # force alignment
    temp ← memory (src);
    if tempa = 16#FF000010#
    then AC.cc ← 2#010#;
    use temp as a received iac message;
    else memory (tempa) ← temp;
    # write operations into memory (tempa) are interpreted
    # as noncacheable
    wait for completion;
    if bad_access
    then AC.cc ← 2#000#;
    else AC.cc ← 2#010#;
    end if;
    end if;

```

**Faults:** STANDARD

**Example:**

```

lda 0xff000010, g7
# g7 ← 0xff000010
synmovq g7, g8
# g8 ← IAC message from address 0xff000010
# AC.cc = 010

```

<b>Opcode:</b>	<b>synmov</b>	600	REG
	<b>synmovl</b>	601	REG
	<b>synmovq</b>	602	REG

**See Also:** synld

tanr, tanrl

Mnemonics: **tanr** Tangent Real  
**tanrl** Tangent Long Real

Format: **tanr\*** *src*, *dst*  
freg/flit freg

**Description:** Calculates the tangent of *src* and stores the result in *dst*. The *src* value is an angle given in radians. The resulting *dst* value is in the range of  $-\infty$  to  $+\infty$ , inclusive; a result of  $-\infty$  or  $+\infty$  will result in a floating invalid-operation exception being signaled.

For the **tanrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the tangent of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src	Dst
$-\infty$	*
-F	-F to +F
-0	-0
+0	+0
+F	-F to +F
$+\infty$	*
NaN	NaN

Notes:  
F Means finite-real number  
\* Indicates floating invalid-operation exception

If the source operand is a finite value, the result will be finite, unless the *src* operand is in a floating-point register and the *dst* operand is not.

**tanr, tanrl**

In the trigonometric instructions, the 80960MC uses a value for  $\pi$  with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 7 titled "Pi" gives this  $\pi$  value, along with some suggestions for representing this value in a program.

**Action:**  $dst \leftarrow \text{tangent}(src);$

**Faults:** **STANDARD** Refer to the discussion of faults at the beginning of this chapter.

**Floating Reserved Encoding** One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	The <i>src</i> operand is $\infty$ .
	One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

**Example:** `tanrl g4, fp0` # tangent of value in g4, g5 is  
# stored in fp0

**Opcode:** **tanr** 68E REG  
**tanrl** 69E REG

**See Also:** **cosr, sinr**

## TEST

<b>Mnemonic:</b>	<b>teste</b>	Test For Equal
	<b>testne</b>	Test For Not Equal
	<b>testl</b>	Test For Less
	<b>testle</b>	Test For Less or Equal
	<b>testg</b>	Test For Greater
	<b>testge</b>	Test For Greater or Equal
	<b>testo</b>	Test For Ordered
	<b>testno</b>	Test For Unordered

<b>Format:</b>	<b>test*</b>	<i>dst</i>
		reg

**Description:** Stores a true (1) in *dst* if the logical AND of the condition code and the mask-part of the opcode is not zero. Otherwise, the instruction stores a false (0) in *dst*.

The following table shows the condition-code mask for each instruction:

Instruction	Mask	Condition
<b>testno</b>	000	Unordered
<b>testg</b>	001	Greater
<b>teste</b>	010	Equal
<b>testge</b>	011	Greater or equal
<b>testl</b>	100	Less
<b>testne</b>	101	Not equal
<b>testle</b>	110	Less or equal
<b>testo</b>	111	Ordered

For the **testno** instruction (Unordered), a true is stored if the condition code is 000<sub>2</sub>; otherwise a false is stored.



# TEST

## Action:

For All Instructions Except **testno**:

```

if (mask and AC.cc) ≠ 2#000#
    then dst ← 1; # dst set for true
    else dst ← 0; # dst set for false
end if;

testno:

```

```

if AC.cc = 2#000#
    then dst ← 1; # dst set for true
    else dst ← 0; # dst set for false
end if;

```

## Faults:

STANDARD

## Example:

```

# assume AC.cc = 100
testl g9 # g9 ← 0x00000001

```

## Opcode:

testno	000	Unordered	22	testno
testl	001	Greater	25	testl
teste	010	Equal	24	teste
testge	011	Greater or equal	26	testge
testl	100	Less	21	testl
testge	101	Not equal	23	testge
testo	110	Less or equal	27	testo
testo	111	Ordered	20	testo

## See Also:

**cmpi, cmpdeci, cmpinci**

**wait**

**Mnemonic:** wait Wait

**Format:** wait src  
reg  
SS

**Description:** Waits on the semaphore. The *src* operand contains the SS of the semaphore.

The processor must be in the supervisor mode to execute this instruction.

The processor checks the semaphore count and the semaphore queue tail. If the count is non-zero and the queue tail is zero, the count is decremented by one and execution of the process continues.

If the count is zero or the queue tail is non-zero, the process is suspended and enqueued on the semaphore.

The process remains queued on the semaphore until it reaches the beginning of the queue and the semaphore receives a **signal** instruction. The process is then dequeued and rescheduled at its dispatching port.

**Action:**

```

x ← atomic_read (semaphore.lock);
if least_significant_bit(x) = 1
  then atomic_write (semaphore.lock) ← x;
  go to wait;
else atomic_write (semaphore.lock) ← x or 1;
  if (semaphore.count = 0) or (semaphore.tail ≠ 0)
    then enqueue process on semaphore;
    x ← atomic_read (semaphore.lock);
    atomic_write (semaphore.lock) ← x xor 1;
    perform process suspension action;
    x ← atomic_read (current_process.lock);
    atomic_write (current_process.lock) ← x xor 1;
    perform process dispatching action;
  else semaphore.count ← semaphore.count - 1;
    x ← atomic_read (semaphore.lock);
    atomic_write (semaphore.lock) ← x xor 1;
  end if;
end if;

```

wait

Faults: STANDARD

Example: wait g8 # waits on semaphore specified in g8

Opcode: wait 669 REG

See Also: condwait, signal

The processor must be in the supervisor mode to execute this instruction. The processor checks the semaphore count and the semaphore queue tail. If the count is non-zero and the queue tail is zero, the count is decremented by one and execution of the process continues. If the count is zero or the queue tail is non-zero, the process is suspended and enqueued on the semaphore. The process remains queued on the semaphore until it reaches the beginning of the queue and the semaphore receives a signal instruction. The process is then dequeued and rescheduled at its dispatching port.

```
end if;
end if;
atomic_write (semaphore.lock) ← x xor 1;
x ← atomic_read (semaphore.lock);
else semaphore.count ← semaphore.count - 1;
perform process dispatching action;
atomic_write (current_process.lock) ← x xor 1;
x ← atomic_read (current_process.lock);
perform process suspension action;
atomic_write (semaphore.lock) ← x xor 1;
x ← atomic_read (semaphore.lock);
then enqueue process on semaphore;
if (semaphore.count = 0) or (semaphore.tail ≠ 0)
else atomic_write (semaphore.lock) ← x or 1;
go to wait;
then atomic_write (semaphore.lock) ← x;
if least_significant_bit(x) = 1
x ← atomic_read (semaphore.lock);
```

Action:

Mnemonic: wait

Format: wait src

Description:

Waits on the semaphore. The src operand contains the 22 of the semaphore.

**xnor, xor**

<b>Mnemonic:</b>	<b>xnor</b>	Exclusive Nor		
	<b>xor</b>	Exclusive Or		
<b>Format:</b>	<b>xnor</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
	<b>xor</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Performs a bitwise XNOR ( <b>xnor</b> instruction) or XOR ( <b>xor</b> instruction) operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
<b>Action:</b>	<b>xnor:</b>	$dst \leftarrow \text{not } (src2 \text{ or } src1) \text{ or } (src2 \text{ and } src1);$		
	<b>xor:</b>	$dst \leftarrow (src2 \text{ or } src1) \text{ and not } (src2 \text{ and } src1);$		
<b>Faults:</b>	STANDARD			
<b>Example:</b>	<pre>xnor r3, r9, r12    # r12 ← r9 XNOR r3 xor g1, g7, g4      # g4 ← g7 XOR g1)</pre>			
<b>Opcode:</b>	<b>xnor</b>	589	REG	
	<b>xor</b>	586	REG	
<b>See Also:</b>	and, andnot, nand, nor, not, notand, notor, or, ornot			



---

*Appendix A*  
*Instruction and Data Structure*  
*Quick Reference*

---



---

# Appendix A Instruction and Data Structure Quick Reference

---

## APPENDIX A

### INSTRUCTION AND DATA STRUCTURE QUICK REFERENCE

This appendix provides quick reference for the 80960MC instructions and data structures.

#### INSTRUCTION QUICK REFERENCE

This section provides two lists of 80960MC instructions: one sorted by assembly-language mnemonic and another sorted by machine-level opcode. In these lists, each entry includes the assembly-language mnemonic for an instruction; the operands (given in the required order); the machine-level opcode and instruction type (i.e., REG, MEM, COBR, CTRL); and the page number in Chapter 17 where the detailed description of the instruction is given.

## Instruction List by Assembler Mnemonic

Mnemonic	Operands			Opcode	Inst. Type	Page
<b>addc</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	5B0	REG	17-6
<b>addi</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	591	REG	17-7
<b>addo</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	590	REG	17-7
<b>addr</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	78F	REG	17-8
<b>addrl</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	79F	REG	17-8
<b>alterbit</b>	<i>bitpos</i> ,	<i>src</i> ,	<i>dst</i>	58F	REG	17-10
<b>and</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	581	REG	17-11
<b>andnot</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	582	REG	17-11
<b>atadd</b>	<i>src/dst</i> ,	<i>src</i> ,	<i>dst</i>	612	REG	17-12
<b>atanr</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	680	REG	17-13
<b>atanrl</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	690	REG	17-13
<b>atmod</b>	<i>src</i> ,	<i>mask</i> ,	<i>src/dst</i>	610	REG	17-15
<b>b</b>	<i>targ</i>			08	CTRL	17-16
<b>bal</b>	<i>targ</i>			0B	CTRL	17-18
<b>balx</b>	<i>targ</i> ,	<i>dst</i>		85	MEM	17-18
<b>bbc</b>	<i>bitpos</i> ,	<i>src</i> ,	<i>targ</i>	30	COBR	17-20
<b>bbs</b>	<i>bitpos</i> ,	<i>src</i> ,	<i>targ</i>	37	COBR	17-20
<b>be</b>	<i>targ</i>			12	CTRL	17-22
<b>bg</b>	<i>targ</i>			11	CTRL	17-22
<b>bge</b>	<i>targ</i>			13	CTRL	17-22
<b>bl</b>	<i>targ</i>			14	CTRL	17-22
<b>ble</b>	<i>targ</i>			16	CTRL	17-22
<b>bne</b>	<i>targ</i>			15	CTRL	17-22
<b>bno</b>	<i>targ</i>			10	CTRL	17-22
<b>bo</b>	<i>targ</i>			11	CTRL	17-22
<b>bx</b>	<i>targ</i>			84	MEM	17-16
<b>call</b>	<i>targ</i>			09	CTRL	17-25
<b>calls</b>	<i>targ</i>			660	REG	17-27
<b>callx</b>	<i>targ</i>			86	MEM	17-29
<b>chkbit</b>	<i>bitpos</i> ,	<i>src</i>		5AE	REG	17-31
<b>classr</b>	<i>src</i>			68F	REG	17-32
<b>classrl</b>	<i>src</i>			69F	REG	17-32
<b>clrbit</b>	<i>bitpos</i> ,	<i>src</i> ,	<i>dst</i>	58C	REG	17-34
<b>cmpdeci</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	5A7	REG	17-36
<b>cmpdeco</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	5A6	REG	17-36
<b>cmpi</b>	<i>src1</i> ,	<i>src2</i>		5A1	REG	17-35
<b>cmpibe</b>	<i>src1</i> ,	<i>src2</i> ,	<i>targ</i>	3A	COBR	17-44
<b>cmpibg</b>	<i>src1</i> ,	<i>src2</i> ,	<i>targ</i>	39	COBR	17-44
<b>cmpibge</b>	<i>src1</i> ,	<i>src2</i> ,	<i>targ</i>	3B	COBR	17-44
<b>cmpibl</b>	<i>src1</i> ,	<i>src2</i> ,	<i>targ</i>	3C	COBR	17-44
<b>cmpible</b>	<i>src1</i> ,	<i>src2</i> ,	<i>targ</i>	3E	COBR	17-44
<b>cmpibne</b>	<i>src1</i> ,	<i>src2</i> ,	<i>targ</i>	3D	COBR	17-44
<b>cmpibno</b>	<i>src1</i> ,	<i>src2</i> ,	<i>targ</i>	38	COBR	17-44
<b>cmpibo</b>	<i>src1</i> ,	<i>src2</i> ,	<i>targ</i>	3F	COBR	17-44

Mnemonic	Operands	Opcode	Inst. Type	Page
cmpinci	src1, src2, dst	5A5	REG	17-37
cmpinco	src1, src2, dst	5A4	REG	17-37
cmpo	src1, src2	5A0	REG	17-35
cmpobe	src1, src2, targ	32	COBR	17-44
cmpobg	src1, src2, targ	31	COBR	17-44
cmpobge	src1, src2, targ	33	COBR	17-44
cmpobl	src1, src2, targ	34	COBR	17-44
cmpoble	src1, src2, targ	36	COBR	17-44
cmpobne	src1, src2, targ	35	COBR	17-44
cmpor	src1, src2	684	REG	17-38
cmporl	src1, src2	694	REG	17-38
cmprr	src1, src2	685	REG	17-40
cmprrl	src1, src2	695	REG	17-40
cmpstr	src1, src2, len	603	REG	17-42
concmpi	src1, src2	5A3	REG	17-47
concmpo	src1, src2	5A2	REG	17-47
condrec	src, dst	646	REG	17-48
condwait	src	668	REG	17-50
cosr	src, dst	68D	REG	17-52
cosrl	src, dst	69D	REG	17-52
cpysre	src1, src2, dst	6E3	REG	17-54
cpysre	src1, src2, dst	6E2	REG	17-54
cvtilr	src, dst	675	REG	17-55
cvtir	src, dst	674	REG	17-55
cvtri	src, dst	6C0	REG	17-56
cvtril	src, dst	6C1	REG	17-56
cvtzri	src, dst	6C2	REG	17-56
cvtzril	src, dst	6C3	REG	17-56
daddc	src1, src2, dst	642	REG	17-58
divi	src1, src2, dst	74B	REG	17-59
divo	src1, src2, dst	70B	REG	17-59
divr	src1, src2, dst	78B	REG	17-60
divrl	src1, src2, dst	79B	REG	17-60
dmovt	src, dst	644	REG	17-62
dsubc	src1, src2, dst	643	REG	17-63
ediv	src1, src2, dst	671	REG	17-64
emul	src1, src2, dst	670	REG	17-65
expr	src, dst	689	REG	17-66
exprl	src, dst	699	REG	17-66
extract	bitpos, len, src/dst	651	REG	17-68
faulte		1A	CTRL	17-69
faultg		19	CTRL	17-69
faultge		1B	CTRL	17-69
faultl		1C	CTRL	17-69
faultle		1E	CTRL	17-69
faultne		1D	CTRL	17-69
faultno		18	CTRL	17-69

Mnemonic	Operands	Opcode	Inst. Type	Page
faulto	REG	1F	CTRL	17-69
fill	dst, value	617	REG	17-71
flushreg	REG	66D	REG	17-72
fmark	COBR	66C	REG	17-73
inspace	src	613	REG	17-74
ld	src, dst	90	MEM	17-75
lda	src, dst	8C	MEM	17-77
ldib	src, dst	C0	MEM	17-75
ldis	src, dst	C8	MEM	17-75
ldl	src, dst	98	MEM	17-75
ldob	src, dst	80	MEM	17-75
ldos	src, dst	88	MEM	17-75
ldphy	src, dst	614	REG	17-78
ldq	src, dst	B0	MEM	17-75
ldt	src, dst	A0	MEM	17-75
ldtime	dst	673	REG	17-79
logbnr	src, dst	68A	REG	17-80
logbnrl	src, dst	69A	REG	17-80
logepr	src1, src2, dst	681	REG	17-82
logeprl	src1, src2, dst	691	REG	17-82
logr	src1, src2, dst	682	REG	17-85
logrl	src1, src2, dst	692	REG	17-85
mark	REG	66B	REG	17-88
modac	mask, src, dst	645	REG	17-89
modi	src1, src2, dst	749	REG	17-90
modify	mask, src, src/dst	650	REG	17-91
modpc	src, mask, src/dst	655	REG	17-92
modtc	mask, src, dst	654	REG	17-94
mov	src, dst	5CC	REG	17-95
movl	src, dst	5DC	REG	17-95
movq	src, dst	5FC	REG	17-95
movqstr	dst, src, len	604	REG	17-96
movr	src, dst	6C9	REG	17-97
movre	src, dst	6E9	REG	17-97
movrl	src, dst	6D9	REG	17-97
movstr	dst, src, len	605	REG	17-99
movt	src, dst	5EC	REG	17-95
muli	src1, src2, dst	741	REG	17-100
mulo	src1, src2, dst	701	REG	17-100
mulr	src1, src2, dst	78C	REG	17-101
mulrl	src1, src2, dst	79C	REG	17-101
nand	src1, src2, dst	58E	REG	17-103
nor	src1, src2, dst	588	REG	17-104
not	src, dst	58A	REG	17-105
notand	src1, src2, dst	584	REG	17-105
notbit	bitpos, src, dst	580	REG	17-106
notor	src1, src2, dst	58D	REG	17-107

Mnemonic	Operands	Opcode	Inst. Type	Page
or	src1, src2, dst	587	REG	17-108
ornot	src1, src2, dst	58B	REG	17-108
receive	src, dst	656	REG	17-109
remi	src1, src2, dst	748	REG	17-111
remo	src1, src2, dst	708	REG	17-111
remr	src1, src2, dst	683	REG	17-112
remrl	src1, src2, dst	693	REG	17-112
resumprcs	src	664	REG	17-115
ret		0A	CTRL	17-116
rotate	len, src, dst	59D	REG	17-118
roundr	src, dst	68B	REG	17-119
roundrl	src, dst	69B	REG	17-119
saveprcs		666	REG	17-120
scaler	src1, src2, dst	677	REG	17-121
scalerl	src1, src2, dst	676	REG	17-121
scanbit	src, dst	641	REG	17-123
scanbyte	src1, src2	5AC	REG	17-124
schedprcs	src	665	REG	17-125
send	dst, src1, src2	662	REG	17-126
sendserv	src	663	REG	17-128
setbit	bitpos, src, dst	583	REG	17-130
shli	len, src, dst	59E	REG	17-131
shlo	len, src, dst	59C	REG	17-131
shrldi	len, src, dst	59A	REG	17-131
shri	len, src, dst	59B	REG	17-131
shro	len, src, dst	598	REG	17-131
signal	dst	66A	REG	17-133
sinr	src, dst	68C	REG	17-134
sinrl	src, dst	69C	REG	17-134
spanbit	src, dst	640	REG	17-136
sqrtr	src, dst	688	REG	17-137
sqrtrl	src, dst	698	REG	17-137
st	src, dst	92	MEM	17-139
stib	src, dst	C2	MEM	17-139
stis	src, dst	CA	MEM	17-139
stl	src, dst	9A	MEM	17-139
stob	src, dst	82	MEM	17-139
stos	src, dst	8A	MEM	17-139
stq	src, dst	B2	MEM	17-139
stt	src, dst	A2	MEM	17-139
subc	src1, src2, dst	5B2	REG	17-141
subi	src1, src2, dst	593	REG	17-142
subo	src1, src2, dst	592	REG	17-142
subr	src1, src2, dst	78D	REG	17-143
subrl	src1, src2, dst	79D	REG	17-143
syncf		66F	REG	17-145
synld	src, dst	615	REG	17-146



A-6

## Instruction List by Opcode

Opcode	Inst. Type	Mnemonic	Operands	Inst. Type	Page
08	CTRL	b	targ	MEM	17-16
09	CTRL	call	targ	MEM	17-25
0A	CTRL	ret		MEM	17-116
0B	CTRL	bal	targ	MEM	17-18
10	CTRL	bno	targ	MEM	17-22
11	CTRL	bg	targ	MEM	17-22
12	CTRL	be	targ	MEM	17-22
13	CTRL	bge	targ	MEM	17-22
14	CTRL	bl	targ	MEM	17-22
15	CTRL	bne	targ	MEM	17-22
16	CTRL	ble	targ	MEM	17-22
17	CTRL	bo	targ	MEM	17-22
18	CTRL	faultno		MEM	17-69
19	CTRL	faultg		MEM	17-69
1A	CTRL	faulte		MEM	17-69
1B	CTRL	faultge		MEM	17-69
1C	CTRL	faultl		MEM	17-69
1D	CTRL	faultne		REG	17-69
1E	CTRL	faultle		REG	17-69
1F	CTRL	faulto		REG	17-69
20	COBR	testno	dst	REG	17-153
21	COBR	testg	dst	REG	17-153
22	COBR	teste	dst	REG	17-153
23	COBR	testge	dst	REG	17-153
24	COBR	testl	dst	REG	17-153
25	COBR	testne	dst	REG	17-153
26	COBR	testle	dst	REG	17-153
27	COBR	testo	dst	REG	17-153
30	COBR	bbc	bitpos, src, targ	REG	17-20
31	COBR	cmpobg	src1, src2, targ	REG	17-16
32	COBR	cmpobe	src1, src2, targ	REG	17-44
33	COBR	cmpobge	src1, src2, targ	REG	17-44
34	COBR	cmpobl	src1, src2, targ	REG	17-44
35	COBR	cmpobne	src1, src2, targ	REG	17-44
36	COBR	cmpoble	src1, src2, targ	REG	17-44
37	COBR	bbs	bitpos, src, targ	REG	17-20
38	COBR	cmpibno	src1, src2, targ	REG	17-44
39	COBR	cmpibg	src1, src2, targ	REG	17-44
3A	COBR	cmpibe	src1, src2, targ	REG	17-44
3B	COBR	cmpibge	src1, src2, targ	REG	17-44
3C	COBR	cmpibl	src1, src2, targ	REG	17-44
3D	COBR	cmpibne	src1, src2, targ	REG	17-44
3E	COBR	cmpible	src1, src2, targ	REG	17-44
3F	COBR	cmpibo	src1, src2, targ	REG	17-44

Opcode	Inst. Type	Mnemonic	Operands	Page
80	MEM	ldob	src, dst	17-75
82	MEM	stob	src, dst	17-139
84	MEM	bx	targ	17-16
85	MEM	balx	targ, dst	17-18
86	MEM	callx	targ	17-29
88	MEM	ldos	src, dst	17-75
8A	MEM	stos	src, dst	17-139
8C	MEM	lda	src, dst	17-77
90	MEM	ld	src, dst	17-75
92	MEM	st	src, dst	17-139
98	MEM	ldl	src, dst	17-75
9A	MEM	stl	src, dst	17-139
A0	MEM	ldt	src, dst	17-75
A2	MEM	stt	src, dst	17-139
B0	MEM	ldq	src, dst	17-75
B2	MEM	stq	src, dst	17-139
C0	MEM	ldib	src, dst	17-75
C2	MEM	stib	src, dst	17-139
C8	MEM	ldis	src, dst	17-75
CA	MEM	stis	src, dst	17-139
580	REG	notbit	bitpos, src, dst	17-106
581	REG	and	src1, src2, dst	17-11
582	REG	andnot	src1, src2, dst	17-11
583	REG	setbit	bitpos, src, dst	17-130
584	REG	notand	src1, src2, dst	17-105
586	REG	xor	src1, src2, dst	17-157
587	REG	or	src1, src2, dst	17-108
588	REG	nor	src1, src2, dst	17-104
589	REG	xnor	src1, src2, dst	17-157
58A	REG	not	src, dst	17-105
58B	REG	ornot	src1, src2, dst	17-108
58C	REG	clrbt	bitpos, src, dst	17-34
58D	REG	notor	src1, src2, dst	17-107
58E	REG	nand	src1, src2, dst	17-103
58F	REG	alterbit	bitpos, src, dst	17-10
590	REG	addo	src1, src2, dst	17-7
591	REG	addi	src1, src2, dst	17-7
592	REG	subo	src1, src2, dst	17-142
593	REG	subi	src1, src2, dst	17-142
598	REG	shro	len, src, dst	17-131
59A	REG	shrdi	len, src, dst	17-131
59B	REG	shri	len, src, dst	17-131
59C	REG	shlo	len, src, dst	17-131
59D	REG	rotate	len, src, dst	17-118
59E	REG	shli	len, src, dst	17-131
5A0	REG	cmpo	src1, src2	17-35
5A1	REG	cmpi	src1, src2	17-35

Opcode	Inst. Type	Mnemonic	Operands	Mnemonic	Inst. Type	Page
5A2	REG	concmpo	src1,	src2	REG	17-47
5A3	REG	concmpi	src1,	src2	REG	17-47
5A4	REG	cmpinco	src1,	src2, dst	REG	17-37
5A5	REG	cmpinci	src1,	src2, dst	REG	17-37
5A6	REG	cmpdeco	src1,	src2, dst	REG	17-36
5A7	REG	cmpdeci	src1,	src2, dst	REG	17-36
5AC	REG	scanbyte	src1,	src2	REG	17-124
5AE	REG	chkbit	bitpos,	src	REG	17-31
5B0	REG	addc	src1,	src2, dst	REG	17-6
5B2	REG	subc	src1,	src2, dst	REG	17-141
5CC	REG	mov	src,	dst	REG	17-95
5DC	REG	movl	src,	dst	REG	17-95
5EC	REG	movt	src,	dst	REG	17-95
5FC	REG	movq	src,	dst	REG	17-95
600	REG	synmov	dst,	src	REG	17-148
601	REG	synmovl	dst,	src	REG	17-148
602	REG	synmovq	dst,	src	REG	17-148
603	REG	cmpstr	src1,	src2, len	REG	17-42
604	REG	movqstr	dst,	src, len	REG	17-96
605	REG	movstr	dst,	src, len	REG	17-99
610	REG	atmod	src,	mask, src/dst	REG	17-15
612	REG	atadd	src1dst,	src, dst	REG	17-12
613	REG	inspacc	src	dst	REG	17-74
614	REG	ldphy	src,	dst	REG	17-78
615	REG	synld	src,	dst	REG	17-146
617	REG	fill	dst	value, len	REG	17-71
640	REG	spanbit	src,	dst	REG	17-136
641	REG	scanbit	src,	dst	REG	17-123
642	REG	daddc	src1,	src2, dst	REG	17-58
643	REG	dsubc	src1,	src2, dst	REG	17-63
644	REG	dmovt	src,	dst	REG	17-62
645	REG	modac	mask,	src, dst	REG	17-89
646	REG	condrec	src,	dst	REG	17-48
650	REG	modify	mask,	src, src/dst	REG	17-91
651	REG	extract	bitpos,	len, src/dst	REG	17-68
654	REG	modtc	mask,	src, dst	REG	17-94
655	REG	modpc	src,	mask, src/dst	REG	17-92
656	REG	receive	src,	dst	REG	17-109
660	REG	calls	targ		REG	17-27
662	REG	send	dst,	src1, src2	REG	17-126
663	REG	sendserv	src		REG	17-128
664	REG	resumpres	src		REG	17-115
665	REG	schedpres	src		REG	17-125
666	REG	savepres	src		REG	17-120
668	REG	condwait	src		REG	17-50
669	REG	wait	src		REG	17-155
66A	REG	signal	dst		REG	17-133

Opcode	Inst. Type	Mnemonic	Operands	Mnemonic	Inst. Type	Page
66B	REG	mark	src1,	concomp	REG	17-88
66C	REG	fmark	src1,	concomp	REG	17-73
66D	REG	flushreg	src1,	compinc	REG	17-72
66F	REG	syncf	src1,	compinc	REG	17-145
670	REG	emul	src1,	compdst	REG	17-65
671	REG	ediv	src1,	compdst	REG	17-64
673	REG	ldtime	dst	scanbyte	REG	17-79
674	REG	cvtir	src,	chkbit	REG	17-55
675	REG	cvtilr	src,	dst	REG	17-55
676	REG	scalerl	src1,	src2,	dst	17-121
677	REG	scaler	src1,	src2,	dst	17-121
680	REG	atanr	src1,	src2,	dst	17-13
681	REG	logepr	src1,	src2,	dst	17-82
682	REG	logr	src1,	src2,	dst	17-85
683	REG	remr	src1,	src2,	dst	17-112
684	REG	cmpor	src1,	src2	REG	17-38
685	REG	cmpr	src1,	src2	REG	17-40
688	REG	sqrtr	src,	dst	REG	17-137
689	REG	expr	src,	dst	REG	17-66
68A	REG	logbnr	src,	dst	REG	17-80
68B	REG	roundr	src,	dst	REG	17-119
68C	REG	sinr	src,	dst	REG	17-134
68D	REG	cosr	src,	dst	REG	17-52
68E	REG	tanr	src,	dst	REG	17-151
68F	REG	classr	src	dst	REG	17-32
690	REG	atanrl	src1,	src2,	dst	17-13
691	REG	logeprl	src1,	src2,	dst	17-82
692	REG	logrl	src1,	src2,	dst	17-85
693	REG	remrl	src1,	src2,	dst	17-112
694	REG	cmporl	src1,	src2	REG	17-38
695	REG	cmprl	src1,	src2	REG	17-40
698	REG	sqrtrl	src,	dst	REG	17-137
699	REG	exprl	src,	dst	REG	17-66
69A	REG	logbnrl	src,	dst	REG	17-80
69B	REG	roundrl	src,	dst	REG	17-119
69C	REG	sinrl	src,	dst	REG	17-134
69D	REG	cosrl	src,	dst	REG	17-52
69E	REG	tanrl	src,	dst	REG	17-151
69F	REG	classrl	src	dst	REG	17-32
6C0	REG	cvtri	src,	dst	REG	17-56
6C1	REG	cvtril	src,	dst	REG	17-56
6C2	REG	cvtzri	src,	dst	REG	17-56
6C3	REG	cvtzril	src,	dst	REG	17-56
6C9	REG	movr	src,	dst	REG	17-97
6D9	REG	movrl	src,	dst	REG	17-97
6E2	REG	cpysre	src1,	src2,	dst	17-54
6E3	REG	cpysre	src1,	src2,	dst	17-54

Opcode	Inst. Type	Mnemonic	Operands			Page
6E9	REG	<b>movre</b>	<i>src</i> ,	<i>dst</i>		17-97
701	REG	<b>mulo</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-100
708	REG	<b>remo</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-111
70B	REG	<b>divo</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-59
741	REG	<b>muli</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-100
748	REG	<b>remi</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-111
749	REG	<b>modi</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-90
74B	REG	<b>divi</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-59
78B	REG	<b>divr</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-60
78C	REG	<b>mulr</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-101
78D	REG	<b>subr</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-143
78F	REG	<b>addr</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-8
79B	REG	<b>divrl</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-60
79C	REG	<b>mulrl</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-101
79D	REG	<b>subrl</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-143
79F	REG	<b>addrl</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>	17-8

Figure A-1: Arithmetic Controls (Chapter 3)



## SUMMARY OF SYSTEM DATA STRUCTURES

The following pages provide a collection of the system data structures presented in this manual. They are grouped by function. The chapter reference below each data structure shows where in this manual this data structure is described.

### Execution Environment

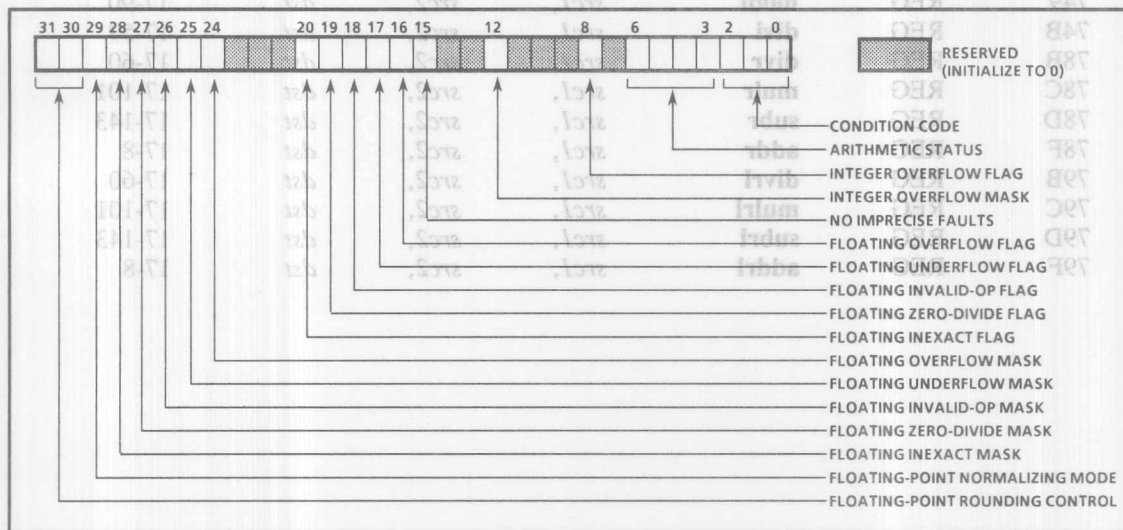


Figure A-1: Arithmetic Controls (Chapter 3)

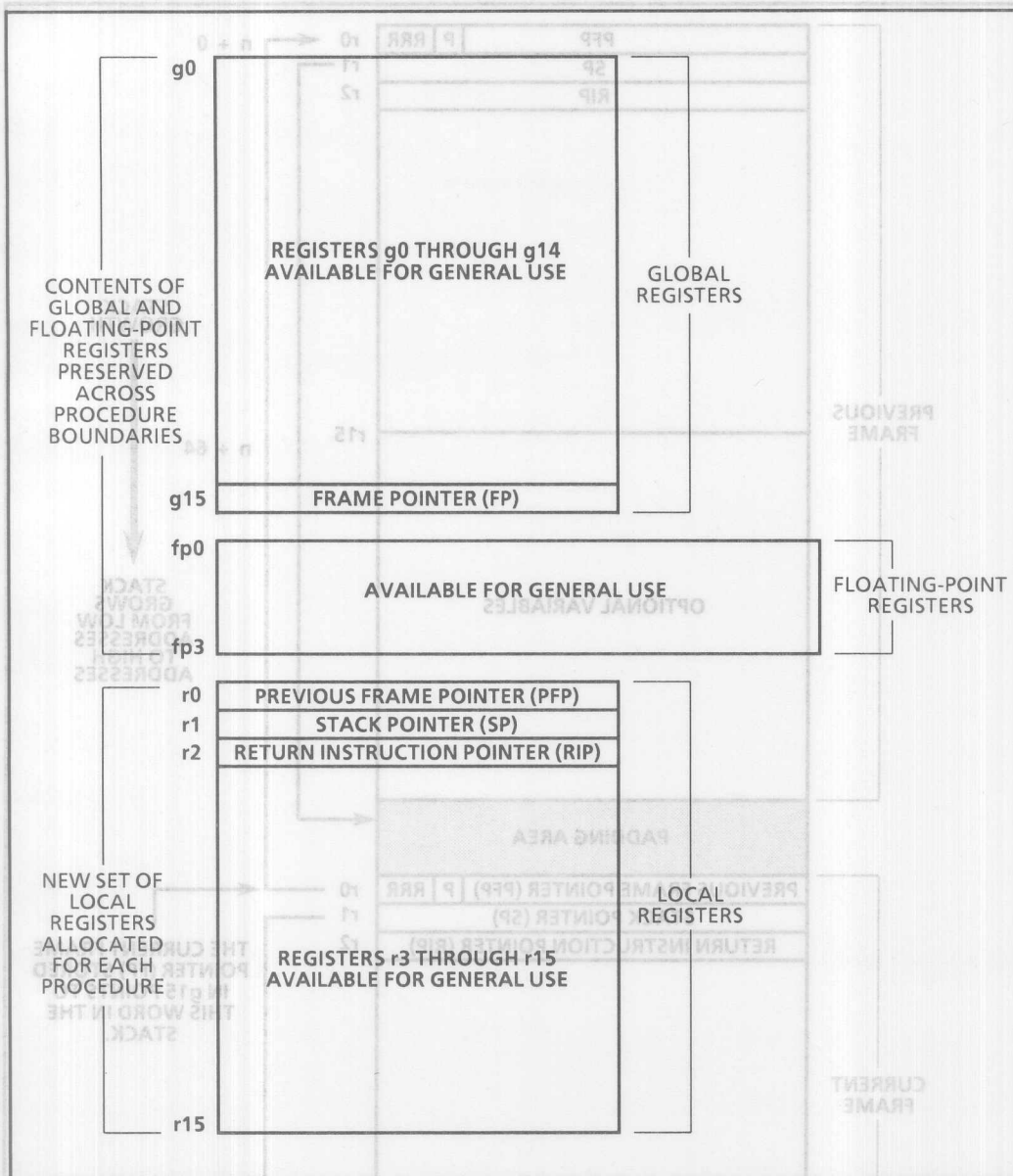


Figure A-2: Registers Available to a Single Procedure (Chapter 3)

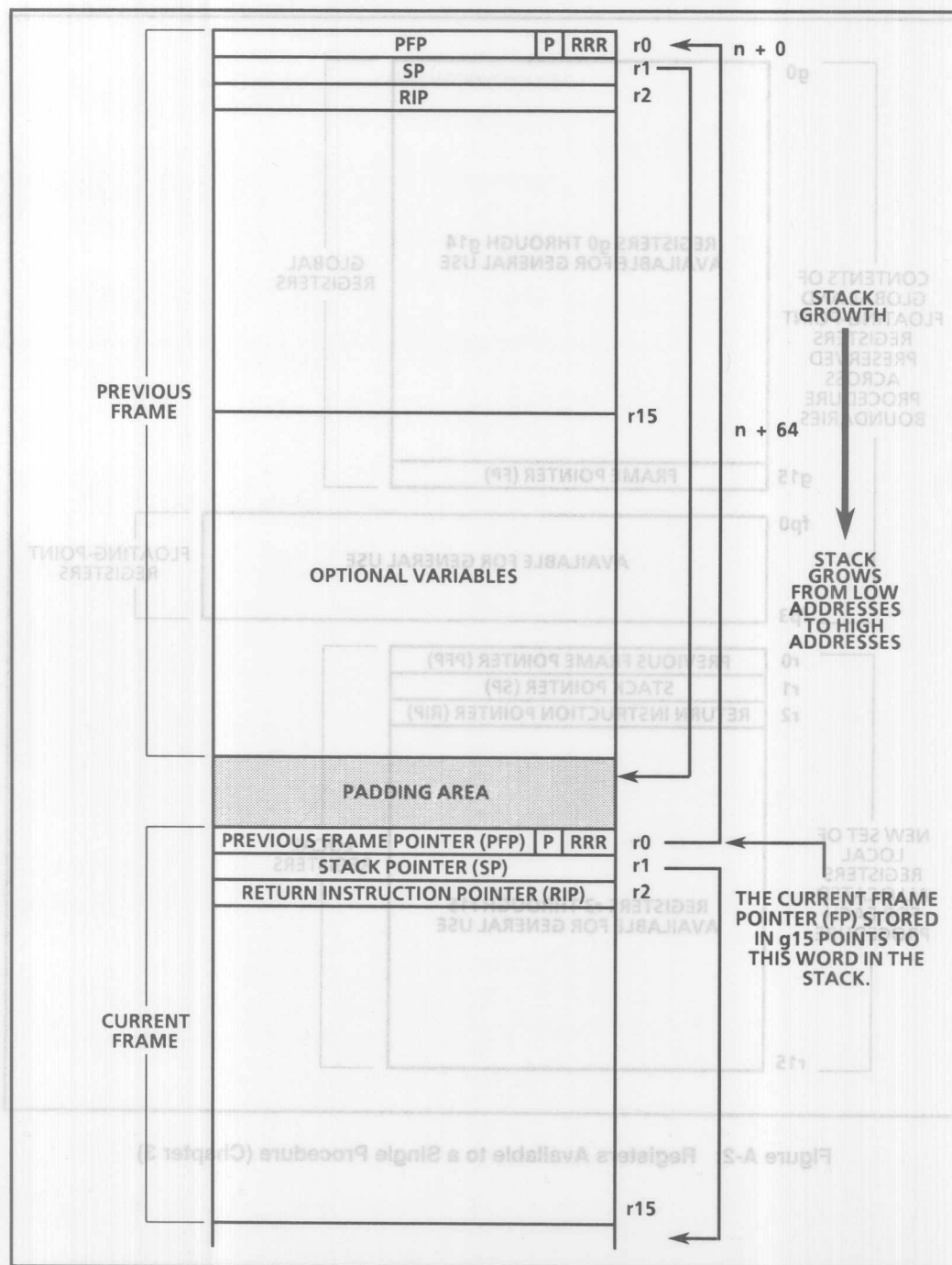


Figure A-3: Procedure Stack Structure (Chapter 4)

## Memory Management

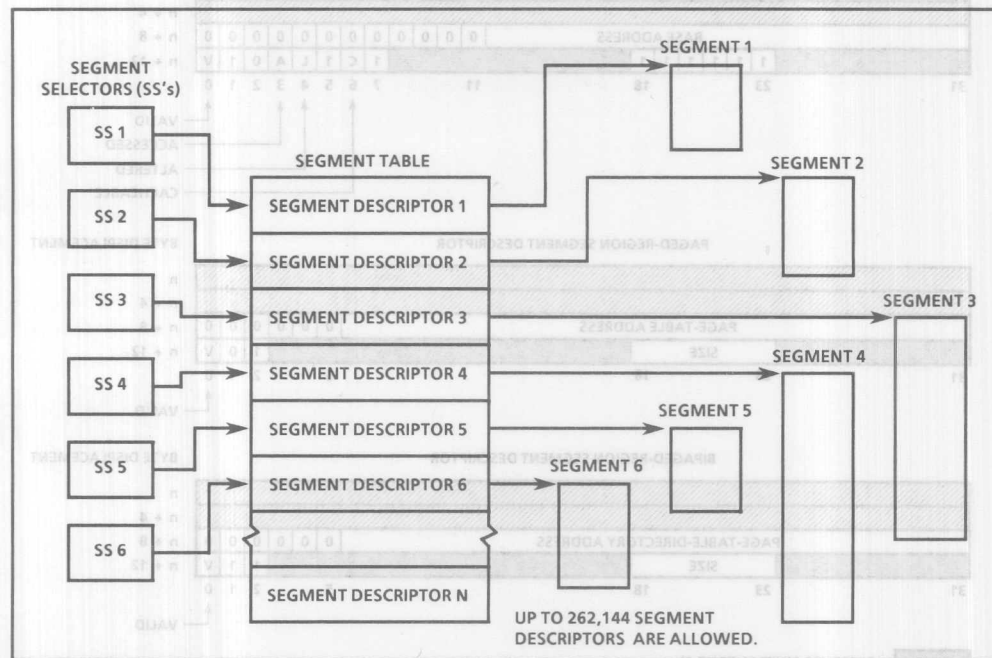


Figure A-4: SS's, Segment Table, and Segments (Chapter 8)

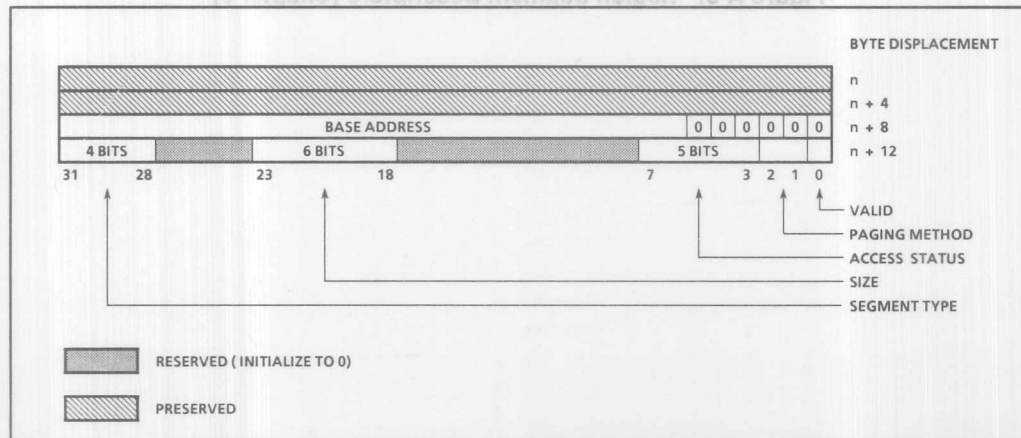


Figure A-5: Generic Segment Descriptor (Chapter 8)

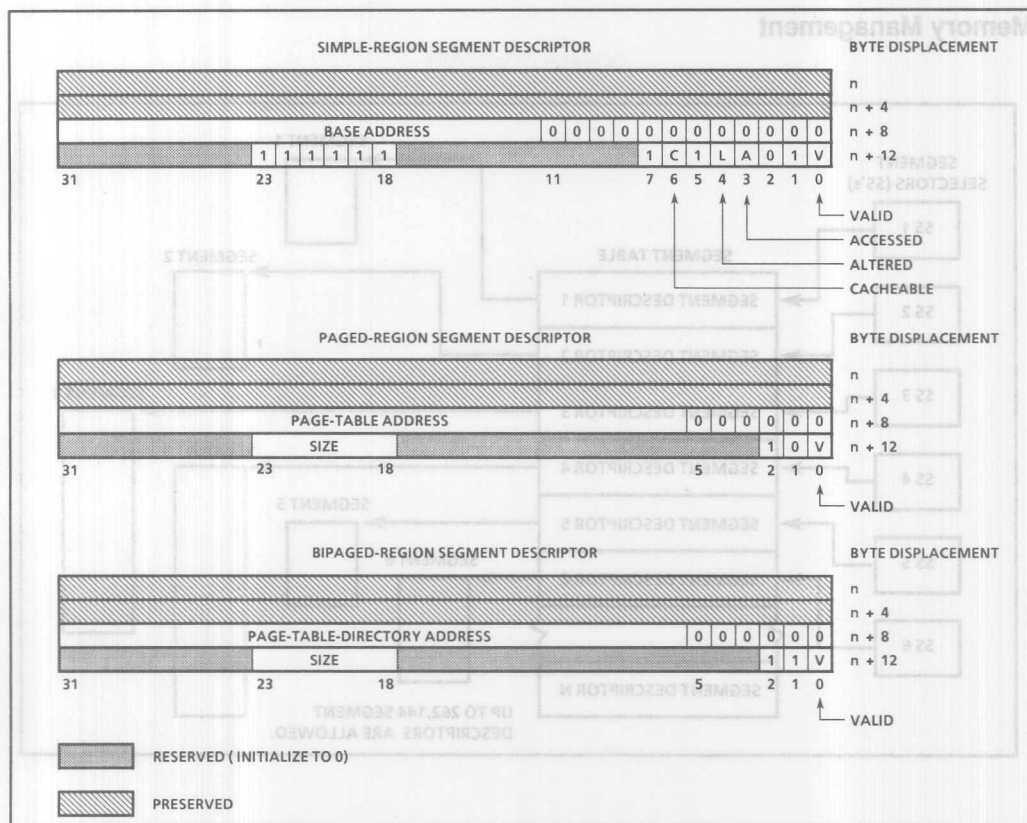


Figure A-4: SS's Segment Table and Segments (Chapter 8)

Figure A-6: Region Segment Descriptors (Chapter 8)

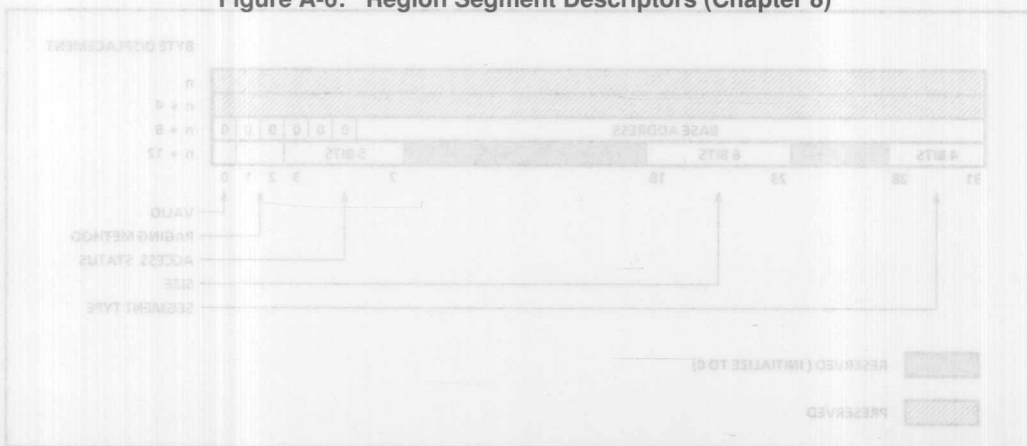


Figure A-5: Generic Segment Descriptor (Chapter 8)

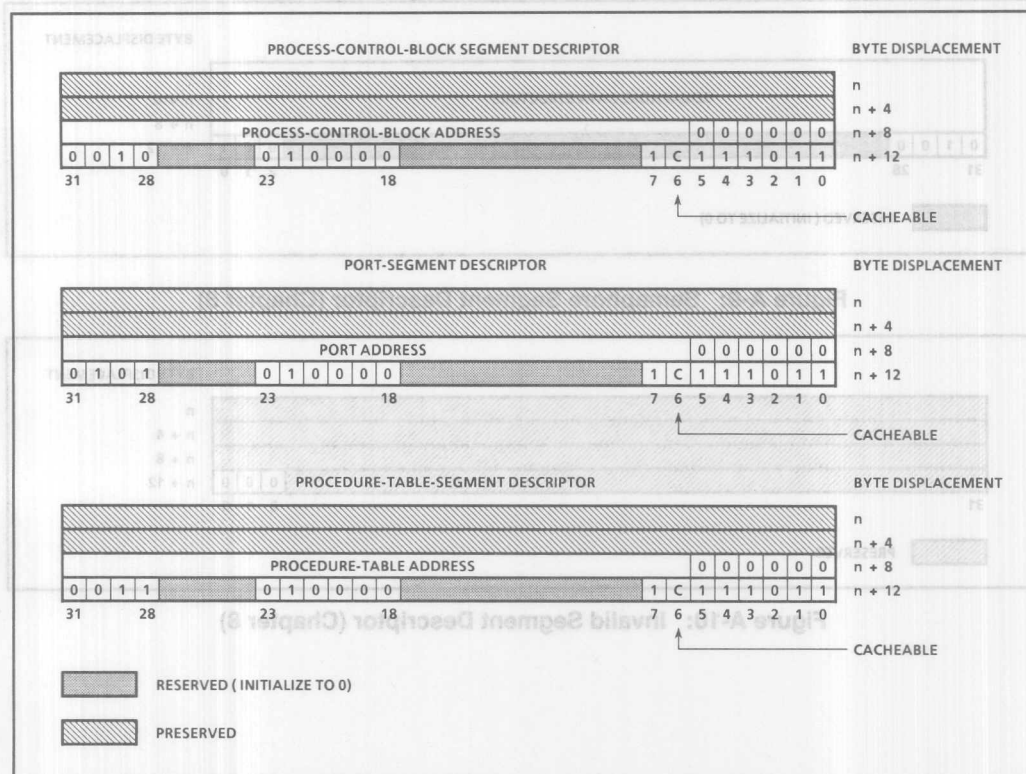


Figure A-7: Process, Port, and Procedure Table Segment Descriptors (Chapter 8)

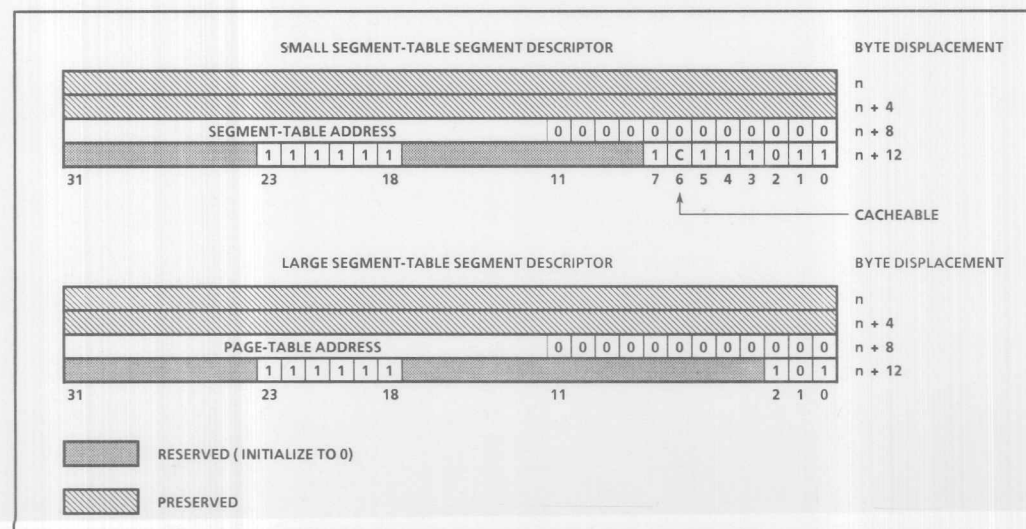


Figure A-8: Segment-Table Segment Descriptors (Chapter 8)



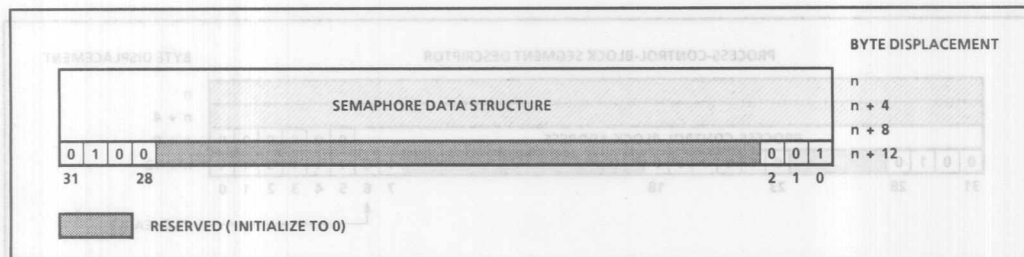


Figure A-9: Semaphore Segment Descriptor (Chapter 8)

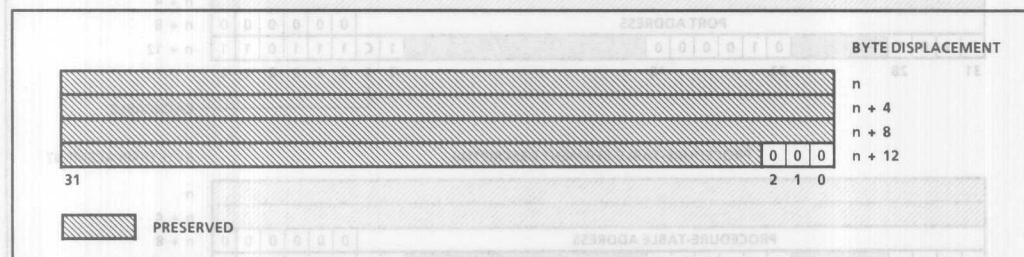


Figure A-10: Invalid Segment Descriptor (Chapter 8)

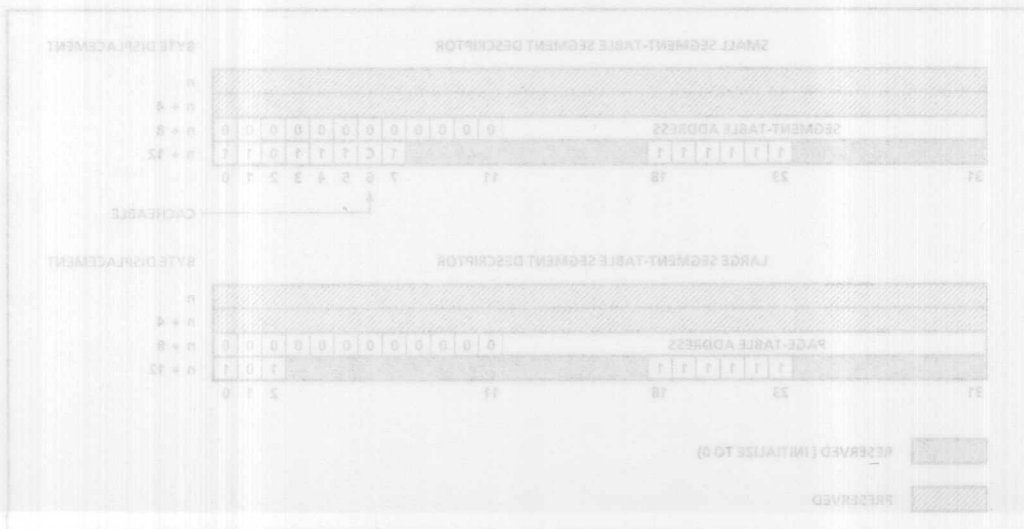


Figure A-8: Segment-Table Segment Descriptors (Chapter 8)

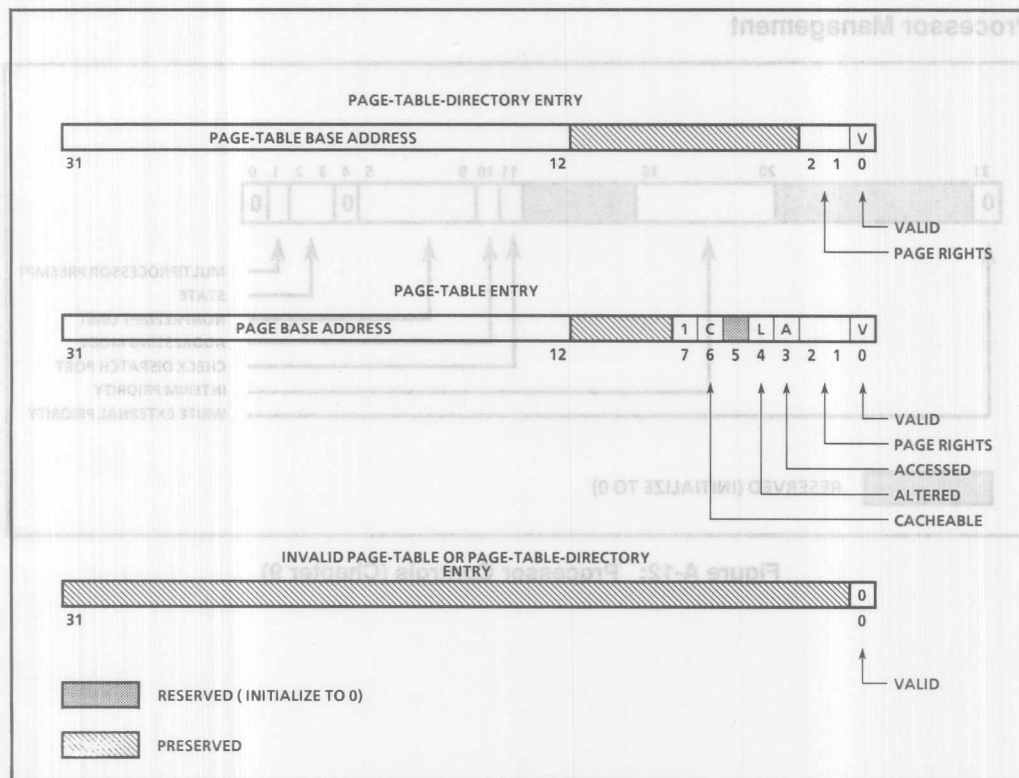


Figure A-11: Page Table or Page-Table Directory Entries (Chapter 8)

Diagram illustrating the structure of the **PRIORITY** register (32 bits).

The register is divided into several fields:

- Reserved (Initialize to 0):** Shaded gray areas in the diagram.
- PAGE RIGHT:** Bit 19.
- PAGE-TABLE ENTRY:** Bits 11-9.
- PAGE-TABLE BASE ADDRESS:** Bit 8.
- MULTIPROCESSOR PREEMPT STATE:** Bit 2.
- NONPREEMPT LIMIT:** Bit 1.
- ADDRESSING MODE:** Bit 0.

Legend: **RESERVED (INITIALIZE TO 0)** (Shaded gray area).

Figure A-12: Processor Controls (Chapter 9)

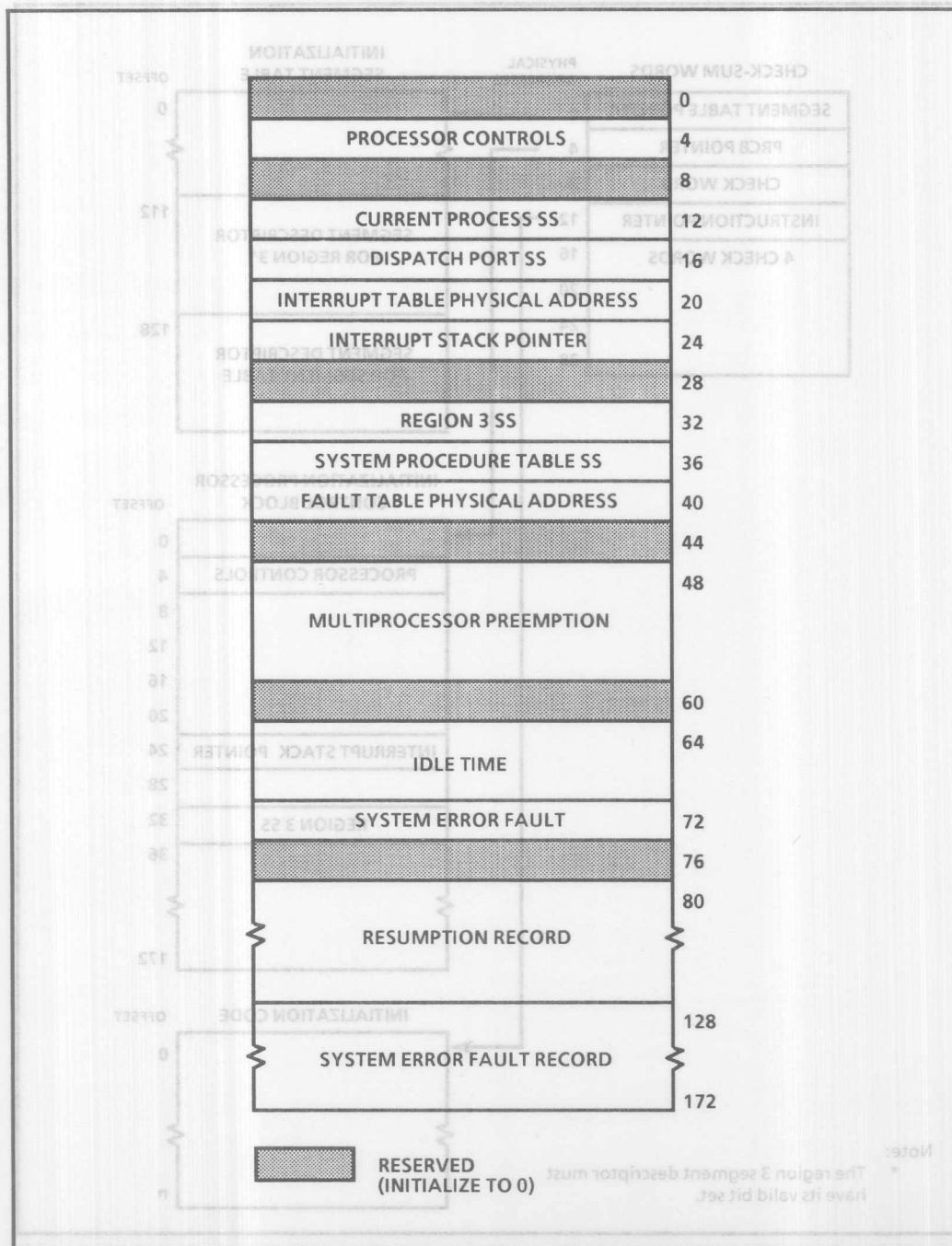


Figure A-13: PRCB (Chapter 9)

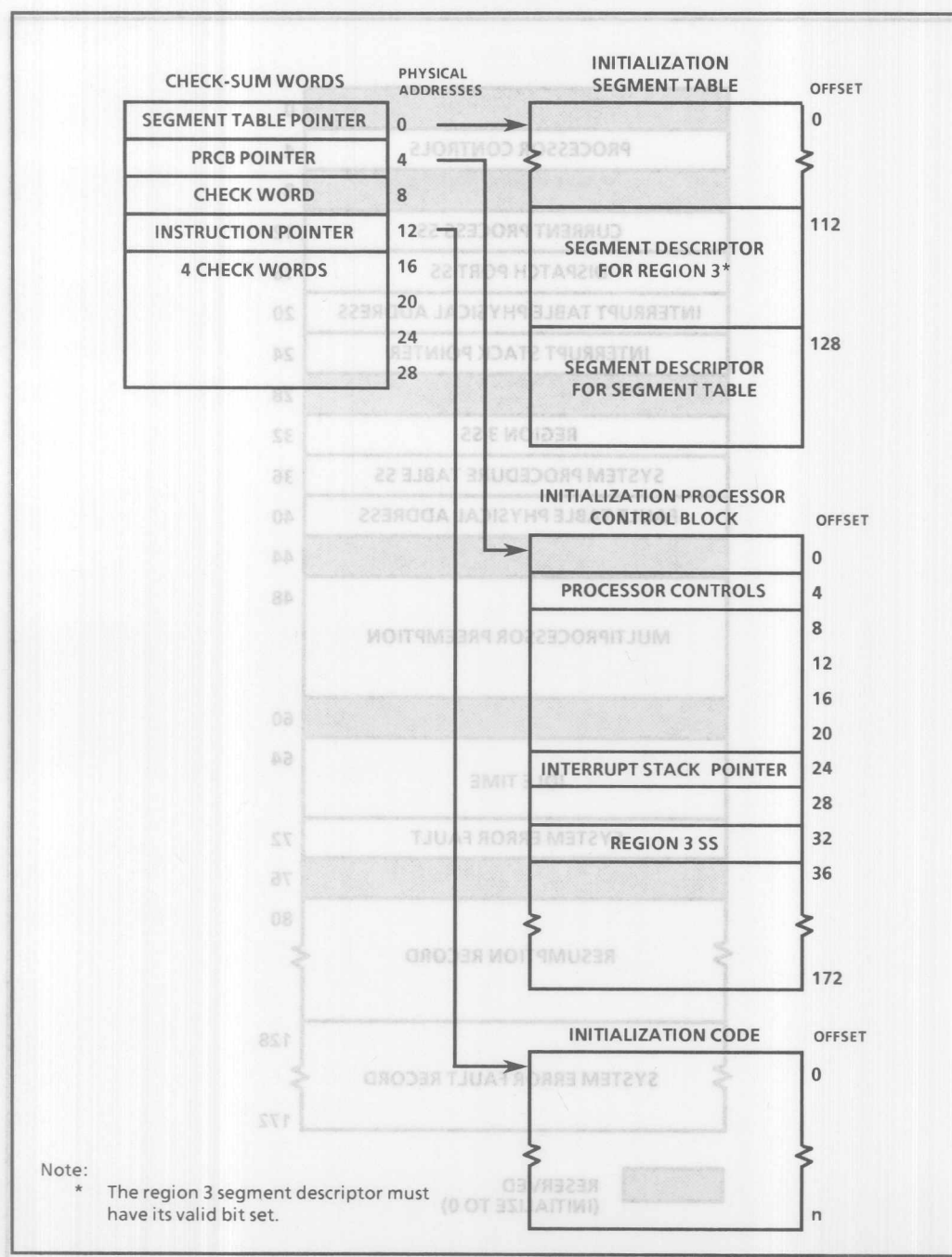


Figure A-14: Initial Memory Image (Chapter 9)

# Interrupt Handling

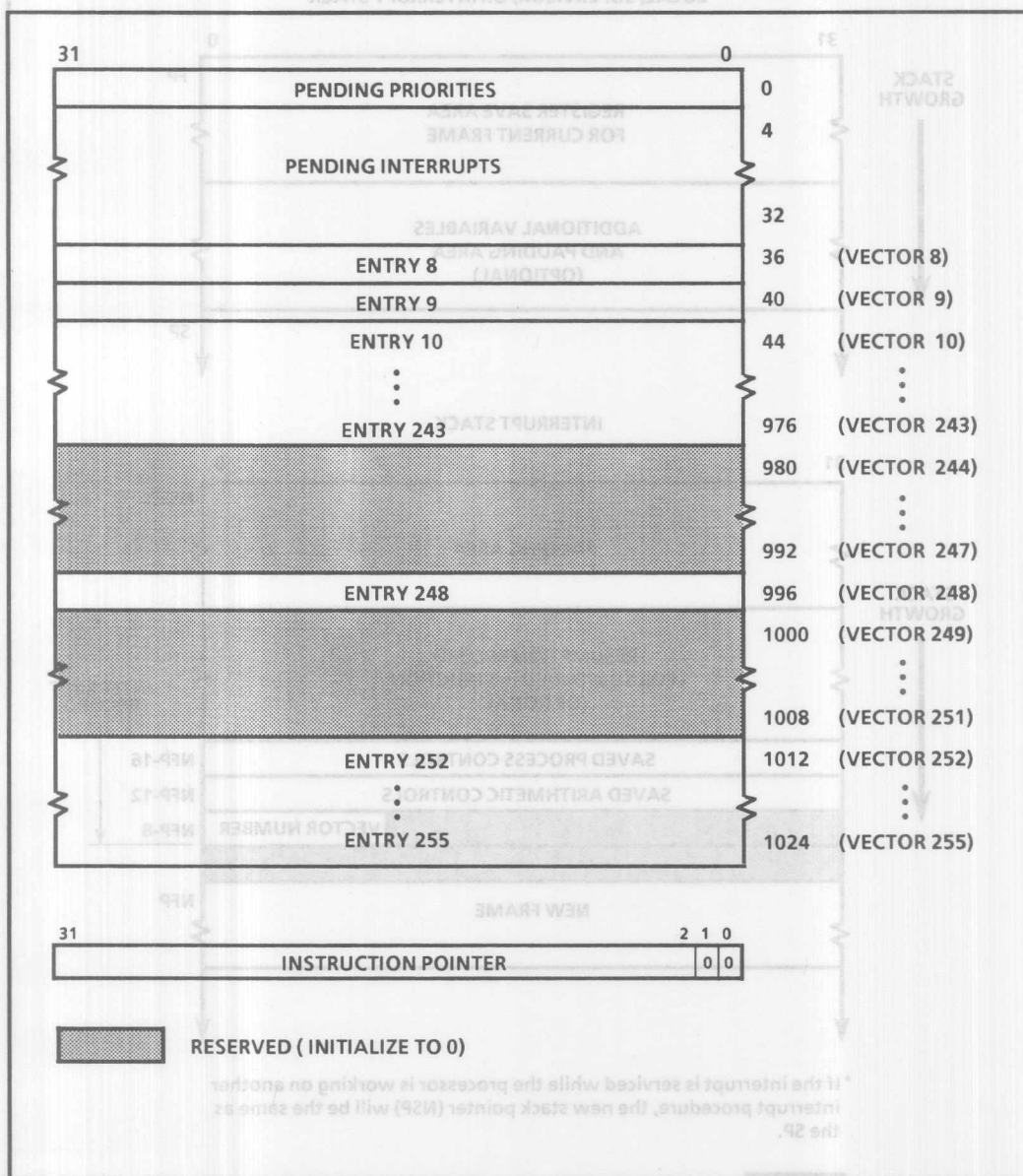


Figure A-15: Interrupt Table (Chapter 10)



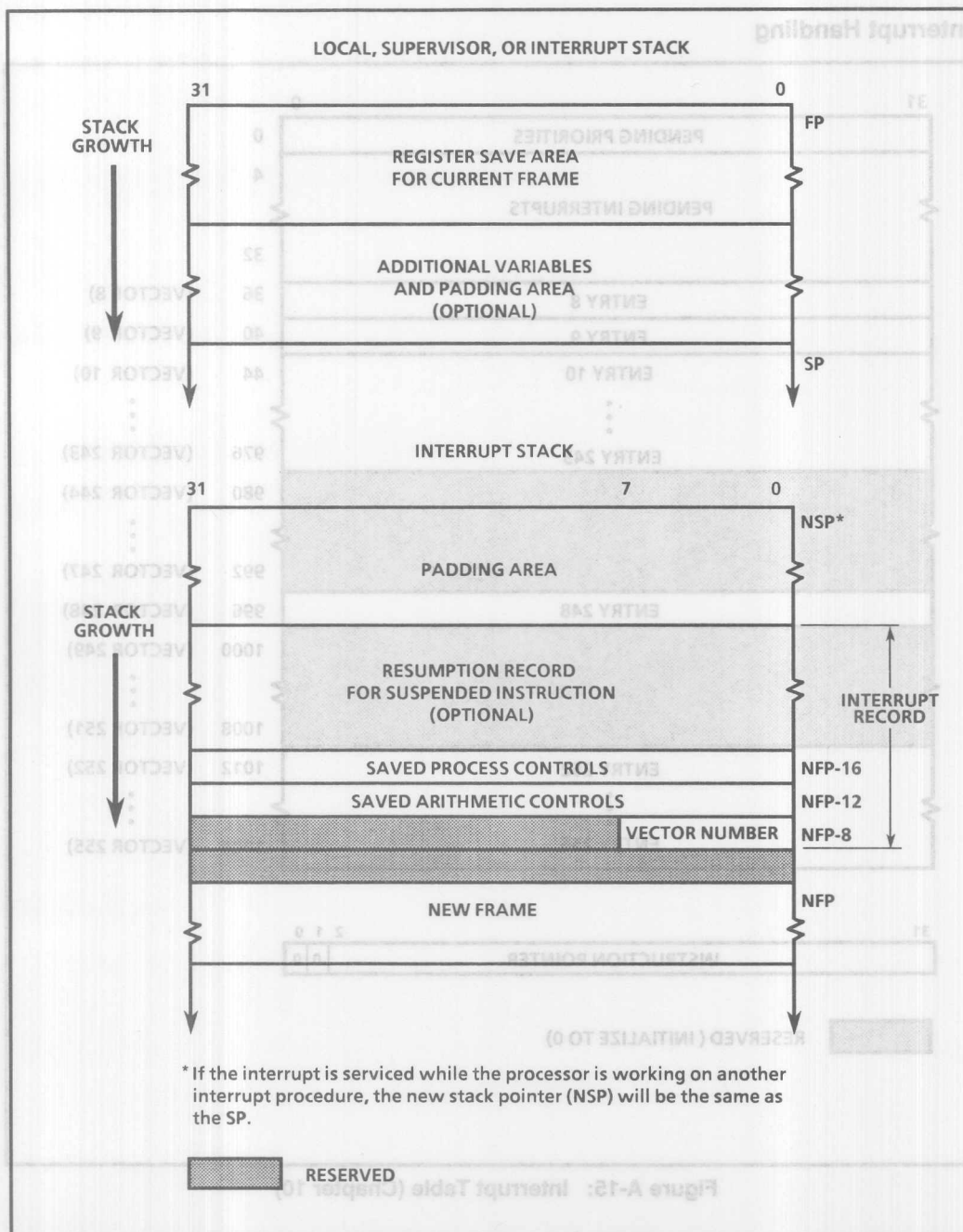


Figure A-16: Interrupt Record on Stack (Chapter 10)

## IACs

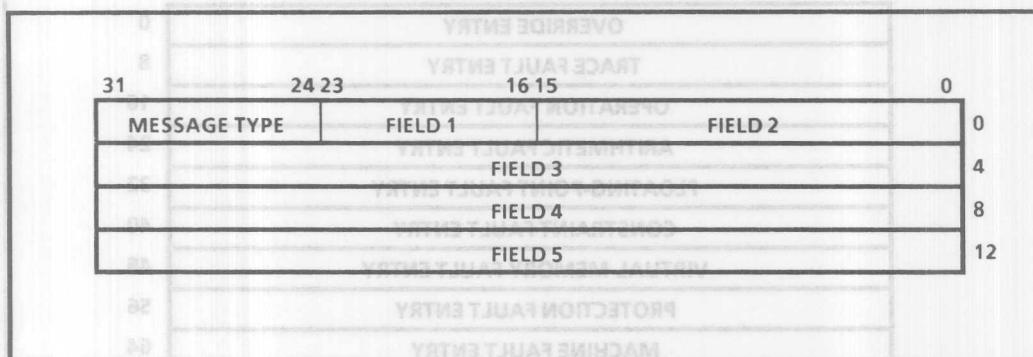


Figure A-17: IAC Message Format (Chapter 11)

## Fault Handling

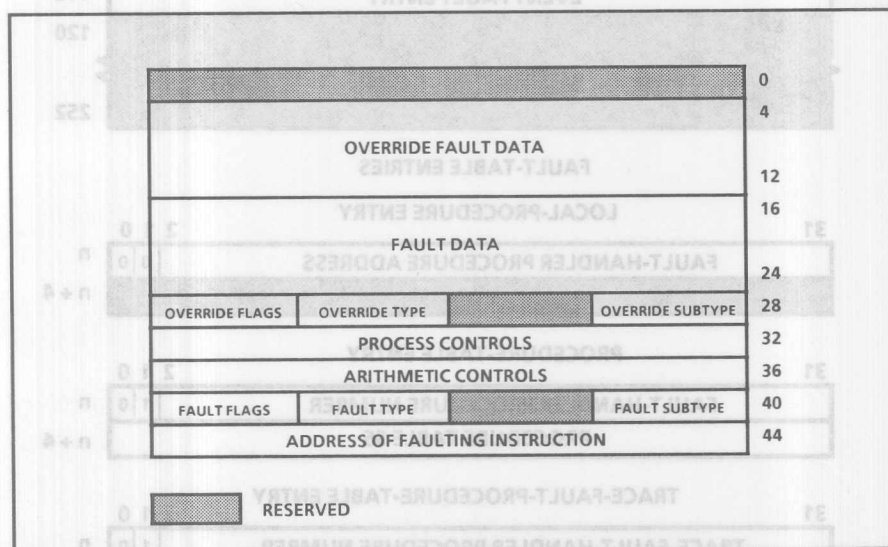


Figure A-18: Fault Record (Chapter 12)

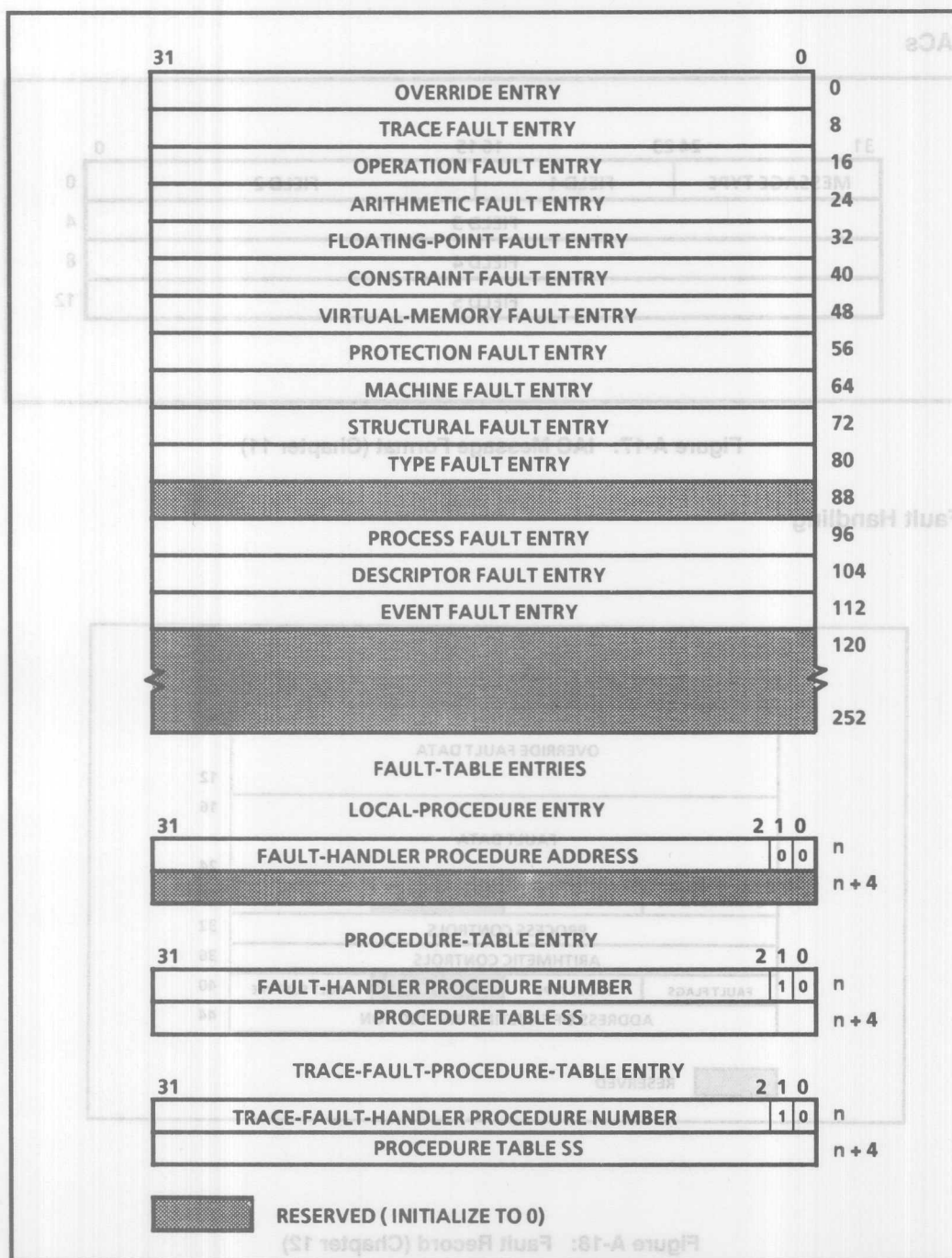


Figure A-19: Fault Table and Fault-Table Entries (Chapter 12)

## Process Management

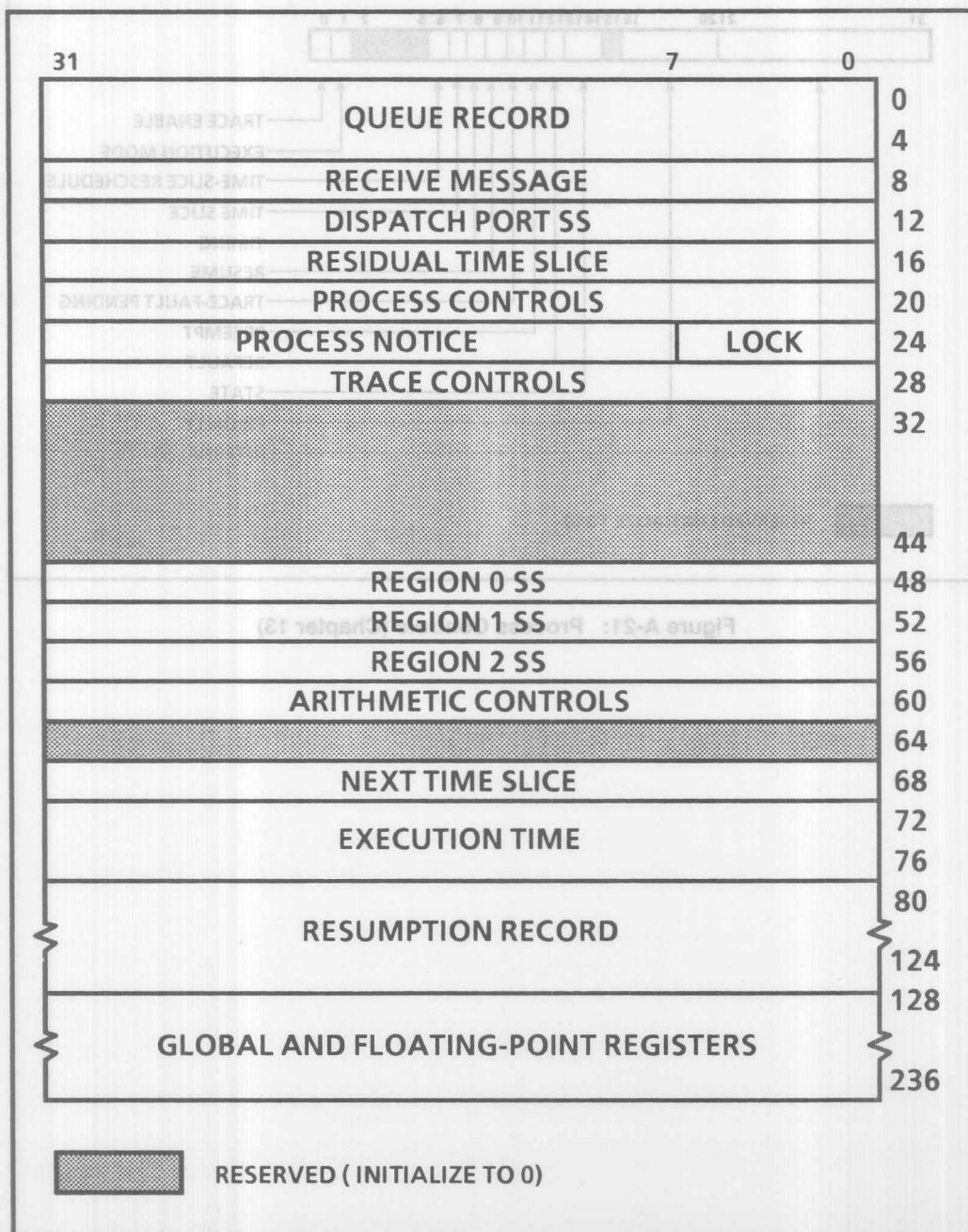


Figure A-20: PCB (Chapter 13)

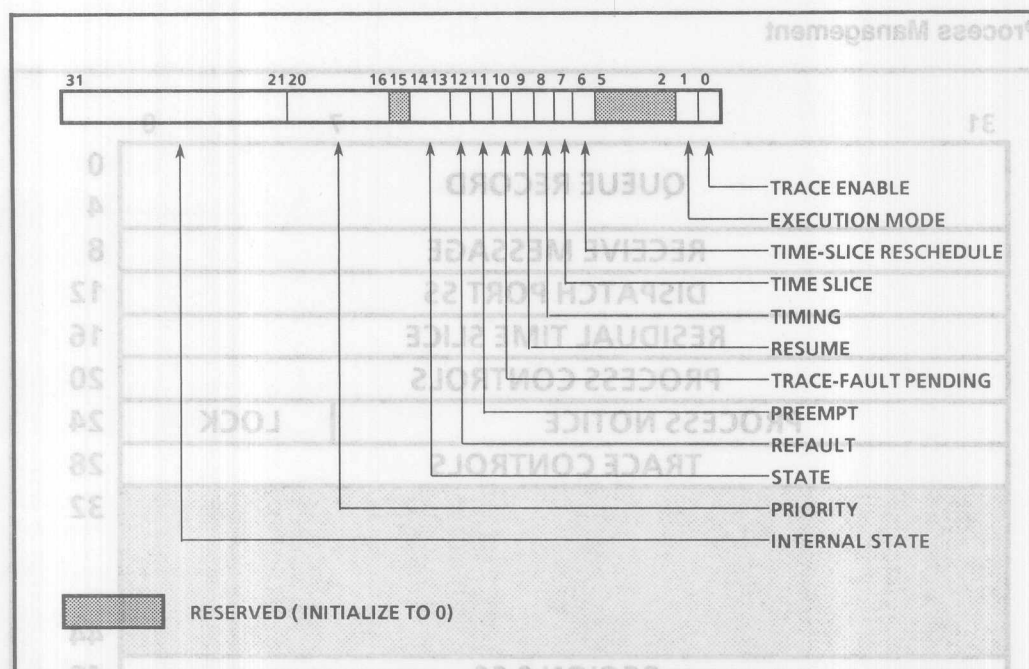


Figure A-21: Process Controls (Chapter 13)

RESERVED (INITIALIZE TO 0)

Figure A-20: PCB (Chapter 13)

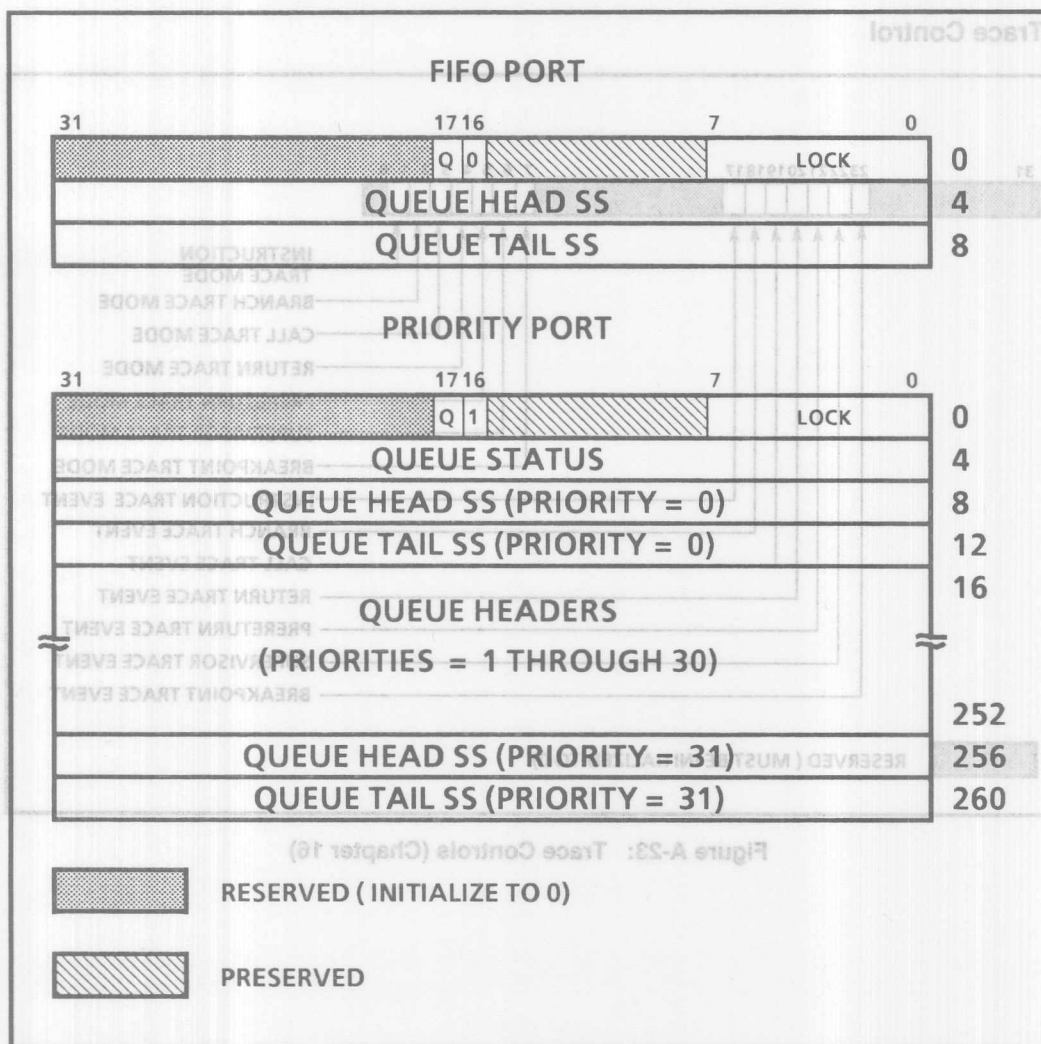


Figure A-22: Ports (Chapter 14)



# Trace Control

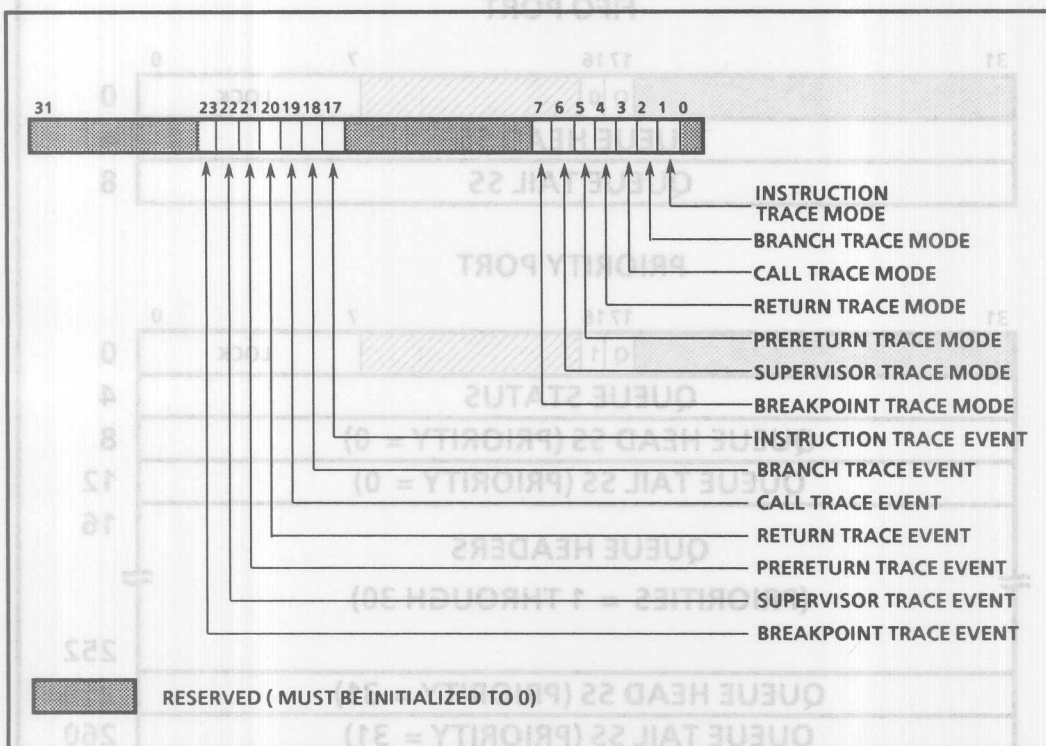


Figure A-23: Trace Controls (Chapter 16)

RESERVED (INITIALIZE TO 0)

RESERVED

Figure A-23: Trace Controls (Chapter 16)

---

*Appendix B*  
*Machine-Level Instruction*  
*Formats*

---

---

# Appendix B Machine-Level Instruction Formats

---

## APPENDIX B

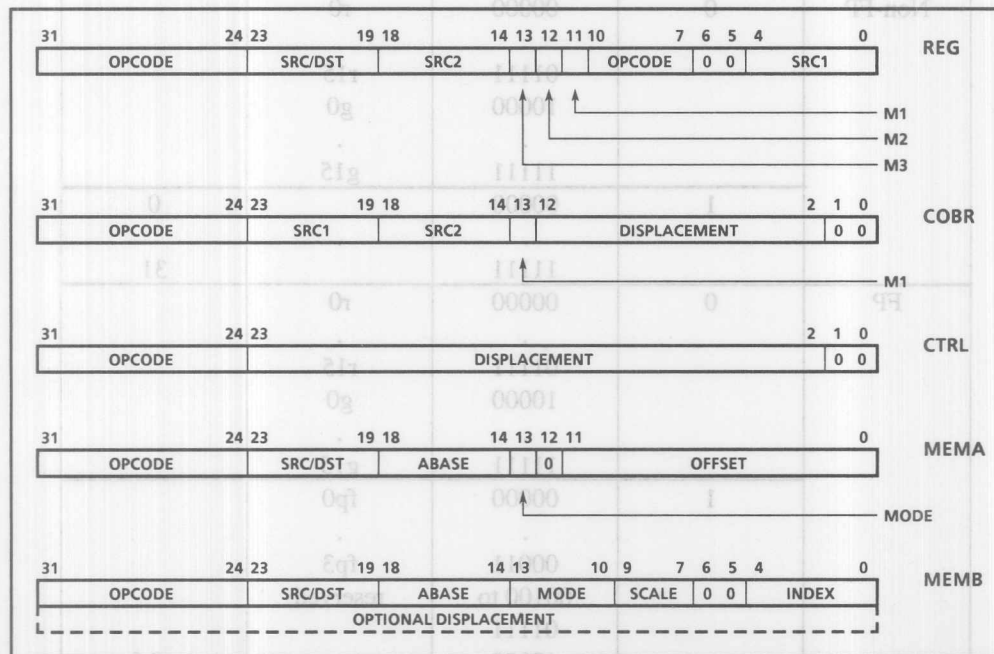
### MACHINE-LEVEL INSTRUCTION FORMATS

This appendix describes the machine-level format for 80960MC instructions. Included is a description of the four instruction formats and how the addressing modes relate to these formats. Also, a table is given that shows the relationship between the machine-level instruction operands and the assembly-language-level instruction operands.

#### GENERAL INSTRUCTION FORMAT

At the machine-level, all the 80960MC instructions are one word long and begin on word boundaries. (One group of instructions allows a second word, which contains a 32-bit displacement.)

There are four basic instruction formats: REG, COBR, CTRL, and MEM. Figure B-1 shows these formats. Each instruction has only one format, which is defined by the opcode field of the instruction.



**Figure B-1: Instruction Formats**

The following sections describe the fields in the instruction word for each format.

## REG FORMAT

The REG format is for operations that are performed on data contained in the global, local, and floating-point registers. The majority of the 80960MC instructions use this format.

The opcode for the REG instructions is 12 bits long (3 hexadecimal digits) and is split between bits 7 through 10 and bits 24 through 31. For example, the opcode for the **addi** instruction is  $591_{16}$ . Here,  $59_{16}$  is contained in bits 24 through 31 and  $1_{16}$  is contained in bits 7 through 10.

The *src1* and *src2* fields specify source operands for the instruction. The operands can be either registers or literals. The mode bits (*m1* for *src1* and *m2* for *src2*) and the instruction type (non-floating point or floating point) determine whether an operand is a register or a literal. Table B-1 shows the relationship between the instruction type, the mode bits, and the *src1* and *src2* operands.

**Table B-1: Encoding of Src1 and Src2 Fields in REG Format**

Inst. Type	M1 or M2	Src1 or Src2 Operand Value	Register Number	Literal Value
Non-FP	0	00000	r0	
		01111	r15	
		10000	g0	
		11111	g15	
	1	00000		0
		11111		31
FP	0	00000	r0	
		01111	r15	
		10000	g0	
		11111	g15	
	1	00000	fp0	
		00011	fp3	
		00100 to 01111	reserved	
		10000		+0.0
		10001 to 10101	reserved	
		10101 to 10110		+1.0
		10111 to 11111	reserved	

For non-floating-point instructions, if a mode bit is set to 0, the respective *src1* or *src2* field specifies a global or local register. If the mode bit is set to 1, the field specifies an ordinal literal in the range of 0 to 31.

For floating-point instructions, if the mode bit is set to 0, the respective *src1* or *src2* field specifies a global or local register (just as it does for non-floating-point instructions). If the mode bit is set to 1, the field specifies either a floating-point register or one of two real-number literals (+0.0 or +1.0). All of the other encoding when the mode bit is set to 1 are reserved. When a reserved encoding is used as a source, the processor either signals an invalid opcode fault or produces an undefined value.

The *src/dst* field can specify either a source operand or a destination operand or both, depending on the instruction. Here again, the mode bit (*m3*) and the instruction type (non-floating point or floating point) determine how this field is used. Table B-2 shows this relationship.

**Table B-2: Encoding of Src/Dst Field in REG Format**

Inst. Type	m3	Src/Dst	Src Only	Dst Only
Non-FP	0	g0 .. g15 r0 .. r15	g0 .. g15 r0 .. r15	g0 .. g15 r0 .. r15
	1	NA	Literal	NA
FP	0	NA	NA	g0 .. g15 r0 .. r15
	1	NA	NA	fp0 .. fp3

Note: NA means not allowed

For non-floating-point instructions, if *M3* is clear, the *src/dst* operand is a global or local register that is encoded as shown in Table B-1. If *M3* is set, the *src/dst* operand can be used only as a *src* operand that is an ordinal literal.

For floating-point instructions, the *src/dst* field is only used to encode destination operands. Here, the encoding is the same as shown in Table B-1, except that the encodings for floating-point literals are not allowed. That is, if *M3* is clear, the destination operand is a global or local register; if *M3* is set, the destination operand is a floating-point register. When a reserved encoding or literal encoding is used as a destination, the processor either signals an invalid opcode fault or produces an undefined result.

## COBR FORMAT

The COBR format is used primarily for control-and-branch instructions. (The test-if instructions also use this format.) The opcode field for this format is 8 bits (two hexadecimal digits).

The *src1* and *src2* fields specify source operands for the instruction. The *src1* field can specify either a global or local register or a literal as determined by mode bit *m1*. (The encoding of the *src1* field is the same as is shown in Table B-1 for the non-floating point instructions.) The *src2* field can only specify a local or global register.



The displacement field contains a signed, two's complement number that specifies a word displacement. The processor uses this value to compute the address of a target instruction that the processor goes to as the result of a comparison. The displacement field can range from  $-2^{10}$  to  $(2^{10} - 1)$ . To determine the IP of the target instruction, the processor converts the displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the IP of the current instruction.

#### NOTE

To allow labels or absolute addresses to be used in the assembly-language version of the COBR format instructions, the Intel 80960MC Assembler converts a *targ* (target) operand value in an assembly-language instruction into the displacement value required for the COBR format, using the following calculation:

$$\text{displacement} = (\text{targ} - \text{IP})/4$$

For the test-if instructions, only the *src1* field is used. Here, this field specifies a destination global or local register (*m1* is ignored).

### CTRL FORMAT

The CTRL format is used for instructions that branch to a new IP, including the branch, branch-if, **bal**, and **call** instructions. The **return** instruction also uses this format. The opcode field for this format is 8 bits (two hexadecimal digits).

The target address for a branch is specified with the displacement field in the same manner as is done with the COBR format instructions. Here, the displacement field specifies a word displacement (also a signed, two's complement number) that can range from  $-2^{21}$  to  $2^{21} - 1$ .

The processor ignores the displacement field for the **return** instruction.

### MEM FORMAT

The MEM format is used for instructions that require a memory address to be computed. These instructions include the load, store, and **lda** instructions. Also, the extended versions of the branch, branch-and-link, and call instructions (**bx**, **balx**, and **callx**) use this format.

There are two MEM formats, MEMA and MEMB. The MEMB format offers the option of including a 32-bit displacement (contained in a second word) to the instruction. Bit 12 of the first word of the instruction determines whether the format is MEMA (clear) or MEMB (set).

For both formats the opcode field is 8 bits long. The *src/dst* field specifies a global or local register. For load instructions, the *src/dst* field specifies the destination register for a word loaded into the processor from memory or, for operands larger than one word, the first of successive destination registers. For store instructions, this field specifies the register or group of registers that contain the source operand to be stored in memory.

The mode bit (or bits for the MEMB format) determine the address mode used for the instruction. Table B-3 summarizes the addressing modes for the two versions of the MEM format. The fields used in these addressing modes are described in the following sections.

Table B-3: Addressing Modes for MEM Format Instructions

Format	Mode Bit(s)	Address Computation
MEMA	0	offset
	1	(abase) + offset
MEMB	0100	(abase)
	0101	(IP) + displacement + 8
	0110	reserved
	0111	(abase) + (index) * $2^{\text{scale}}$
	1100	displacement
	1101	(abase) + displacement
	1110	(index) * $2^{\text{scale}}$ + displacement
	1111	(abase) + (index) * $2^{\text{scale}}$ + displacement

Note:

1. In the address computations above, a field in parentheses (e.g., (abase)) indicates that the value in the specified register is used in the computation.
2. The use of a reserved encoding causes an invalid opcode fault to be signaled.

### MEMA Format Addressing

The MEMA format provides two addressing modes:

- absolute offset
- register indirect with offset

The offset field specifies an unsigned byte offset from 0 to 4096. The abase field specifies a global or local register that contains an address in memory. The address is interpreted as either a virtual address or a physical address depending on whether the processor is operating in virtual-addressing or physical-addressing mode, respectively.

For the absolute offset addressing mode (the mode bit is clear), the processor interprets the offset field as an offset from byte 0 of the current process address space. The abase field is ignored. Using this addressing mode along with the **lda** instruction allows a constant of from 0 to 4096 to be loaded into a register.

For the register indirect with offset addressing mode (the *mode* bit is set), the value in the *offset* field is added to the address in the abase register. Setting the offset value to zero creates a register indirect addressing mode; however, this operation can generally be carried out faster by using the MEMB version of this addressing mode.

## MEMB Format Addressing

The MEMB format provides the following seven addressing modes:

- absolute displacement
- register indirect
- register indirect with displacement
- register indirect with index
- register indirect with index and displacement
- index with displacement
- IP with displacement

The abase and index fields specify local or global registers, the contents of which are used in the address computation. When the index field is used in an addressing mode, the processor automatically scales the value in the index register by the amount specified in the scale field. Table B-4 gives the encoding of the scale field. The optional displacement field is contained in the word following the instruction word. The displacement is a 32-bit, signed, two's complement value.

Table B-4: Encoding of Scale Field

Scale	Scale Factor (Multiplier)
000	1
001	2
010	4
011	8
100	16
101 to 111	reserved

Note:

The use of a reserved encoding causes an invalid opcode fault to be signaled.

For the IP with displacement mode, the value of the displacement field plus 8 is added to the address of the current instruction.

---

# *Appendix C*

## *Instruction Timing*

---

---

# Appendix C Instruction Timing

---

## APPENDIX C INSTRUCTION TIMING

This appendix describes the 80960MC processor's instruction pipeline and how it affects the timing of instructions. The number of clock cycles required for each instruction are also given here.

### INTRODUCTION

The 80960 architecture defines several mechanisms for increasing processor performance through the use of pipelining and parallel execution of instructions. This appendix describes how these mechanisms have been incorporated into the design of the 80960MC processor and provides information to help programmers maximize the performance of the processor.

### INTERNAL STRUCTURE OF THE 80960MC PROCESSOR

The 80960MC processor is composed of the following six major functional units (shown in Figure C-1):

- Memory Management Unit
- Bus Control Logic
- Instruction Fetch Unit and Instruction Cache
- Instruction Decoder
- Micro-Instruction Sequencer and ROM
- Instruction Execution Unit
- Floating Point Unit

These units function independently from one another, but in close cooperation. The functions of each of these units is described in the following sections.

#### Memory Management Unit

The Memory Management Unit (MMU) translates virtual addresses into physical addresses and sends the resulting address to the Bus Control Logic (BCL). When the processor is in the physical addressing mode, the MMU is effectively bypassed and addresses are passed directly to the Bus Control Logic (BCL). The MMU becomes active in address translation, in the following situations:

- When the virtual addressing mode is used.
- When the processor accesses system data structures (such as the PRCB, dispatch ports, PCBs, etc.) as part of high-level primitive operations like dispatching a process or sending a signal to a semaphore.



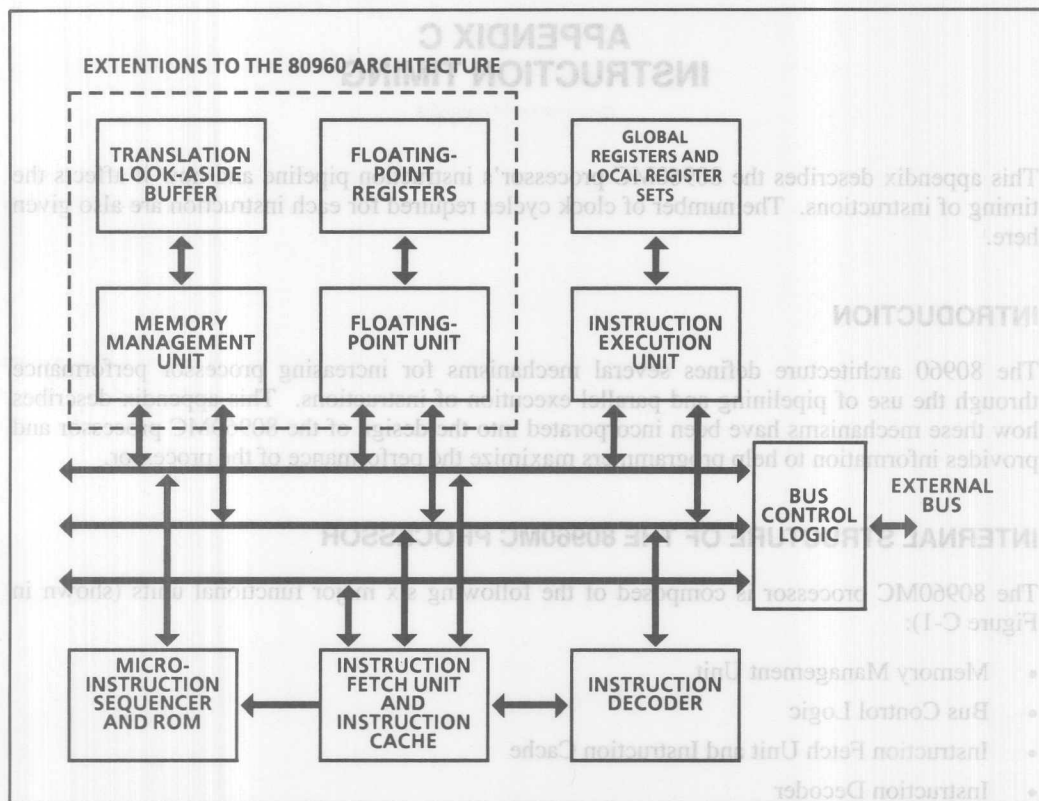


Figure C-1: Block Diagram of the 80960MC Processor

To streamline the address translation process, the MMU maintains a 44 entry cache on the chip called the translation lookaside buffer (TLB). This cache is used to store often-used addresses that have already been translated. The first 12 entries in the TLB hold addresses for system defined data structures such as pointers to the page tables for the four regions of the address space for the current process. The next 32 entries contain pointers to 32 pages currently mapped into the physical address space. These 32 entries point to 128K bytes of memory, which yields a cache hit ratio of 98% for typical applications.

The MMU is also able to perform type checks when referencing certain types of system data structures (such as PCBs, ports, or procedure tables), while instructions are being executed in other parts of the processor. Type checking is thus often overlapped with other processor activities.

### Bus Control Logic

The BCL provides the interface between the processor and the external world. This interface consists of a multiplexed, burst bus, which is capable of memory-access rates of over 53 Megabytes/second (with a 20 Mhz CPU clock). The BCL accepts requests from the MMU, prioritizes them, and executes them. It attempts to maximize bus access efficiency through buffering and burst accesses.

The BCL provides a queuing mechanism that can buffer up to three outstanding requests at any given time. This mechanism, coupled with other 80960MC features (such as scoreboarding, which is discussed later), allow other units in the 80960MC to continue operation without waiting for bus requests to be completed. As a result, the execution of most memory reference instructions require little or no delay in the instruction execution pipeline.

The BCL generates burst cycles on the external bus, which allow from one to 16 bytes of data to be read or written in a single operation. The processor takes advantage of burst transfers in several ways. First, multiple-register load or store operations can be carried out in a single bus operation, using the **ldl** (load long), **ldt** (load triple), and **ldq** (load quad) instructions and the corresponding **stl** (store long), **stt** (store triple), and **stq** (store quad) instructions. Second, instructions can be fetched in 16-byte bursts, thereby reducing bus traffic for instruction fetches. Third, floating-point values of 32, 64 or 80 bits can be stored in a single bus operation. Fourth, the reading and writing of system data structures as part of process management tasks (such as switching processes or sending messages) can be carried out at very fast rates.

### Instruction Fetch Unit and Instruction Cache

The Instruction Fetch Unit (IFU) acts as an intelligent "buffer" for the Instruction Decoder (ID). Its purpose is to present the instruction stream to the ID in the fastest and most transparent way possible. The IFU uses several mechanisms to accomplish this goal, as described in the following paragraphs.

The IFU maintains a 512 byte, direct-mapped instruction cache. This cache allows very fast access to instructions. While the other units in the processor are executing instructions, the IFU looks ahead in flow of instructions stored in the instruction cache. If a cache miss is detected (that is, an instruction that will soon be needed is not in the instruction cache), the IFU issues a prefetch request to the MMU. Upon receiving the requested instruction, the IFU updates the instruction cache. In most cases, this fetch and load will take place before the ID requires the instruction. The major exception to this rule happens on branch conditions.

The IFU works closely with the ID in handling branch conditions. The ID informs the IFU of any branch operations that are about to take place. Such notifications take place on unconditional branches and on conditional branches in which the condition code is valid. When the IFU is notified of a branch, it checks for a cache hit on the desired instruction. If the instruction is not present, the IFU begins fetching instructions for the new control path.

To further minimize delays in the instruction pipeline, the ID sends a special signal to the IFU whenever instructions are required immediately. The IFU then passes the fetched instructions to the ID directly, rather than writing them to the cache and reading them back out again. This technique is called an instruction-cache bypassing.

The instruction pointer (IP) register in the processor and the IFU maintain several instruction pointers. These pointers point to instructions at various stages of the fetch-decode-execute pipeline. If a fault is signaled from any unit, the processor uses these pointers to determine the problem and preserve the state of the processor.

## Instruction Decoder

The ID decodes the instructions it receives from the IFU and routes them to the appropriate execution units. In doing this, it attempts to keep the computing resources of the processor working at the highest possible levels.

Instructions are decoded into the following four groups, according to how the instructions are executed:

- Simple Instructions
- Floating Point and Branch Instructions
- Complex Instructions
- Load and Store Instructions

The following paragraphs list the instructions in each of these groups and describe how the ID handles them.

## Simple Instructions

The instructions in the simple-instruction group require very little decoding. These instructions include logical; comparison; shift; integer add and subtract; and ordinal add and subtract instructions. The ID decodes these instructions and passes them to the instruction execution unit (IEU), where they are executed, usually in a single clock period.

## Floating Point and Branch Instructions

All floating-point instructions are executed by the floating-point unit (FPU). Often, the execution of floating-point instructions requires interaction between the FPU, ID, and Micro-Instruction Sequencer (MIS). For example, the FPU may require access to the general-purpose registers (maintained by the IEU). Here, the ID assists in supplying data to the FPU. Also, many of the floating-point instructions are executed by means of microcode. The FPU gets the microcode from the MIS.

The ID executes branch instructions directly. If the branches are unconditional, no interaction with the processor's other execution units is required.

On conditional branch instructions, the ID uses a condition code scoreboard to streamline the branching process. Scoreboarding is a mechanism by which various resources within the processor can be marked as *in use* (or *pending a result*). When one of the execution units in the processor is in the process of altering the condition code, it marks the condition code scoreboard. When the ID prepares to execute a conditional branch instruction, it checks the condition code scoreboard. If the scoreboard is marked as *in use*, the ID waits for the result before proceeding. If the condition code scoreboard is clear, the ID signals the IFU immediately if a change in program flow is about to happen.

Conditional fault instructions (fault-if instructions) are also executed in the ID. These operations differ from conditional branches in that they result in a fault event being generated, followed by an implicit call to the appropriate fault-handler routine.

As a result of the pipelining described above, branches can often be carried out in zero clock cycles. For example, the branch instruction (**b**) shown below will execute in zero cycles, since the branch time is overlapped completely by the execution time of the floating-point instruction (**sinr**).

```
sinr    g0, g1
b       some_location
.
```

```
some_location:
mov     g1, g2
```

The branch-if instruction (**be**) in the following example is also executed in zero cycles:

```
cmp     0x10, r9
divi    r10, r11, r10
be      go_here
.
```

```
go_here:
mov     g1, g2
```

Here, the comparison instruction (**cmp**) is placed early in the instruction stream, allowing the branch condition based on the value of r9 to take place while the integer divide instruction (**divi**) is being executed.

## Complex Instructions

Complex instructions are those that are executed using one or more microcode instructions. Examples of such instructions are the **flushreg** (flush local registers), **mark**, and **fmark** (force mark) instructions. The ID decodes complex instructions and forwards them to the MIS unit. The MIS then sends the equivalent microcode to the IEU.

## Load and Store Instructions

Load and store instructions are those that request data to be read from or written into memory. The ID sends these instructions directly to the MMU and BCL, which executes them.

The ID is responsible for converting the addressing information encoded in load, store, branch, and call instructions into an effective memory addresses. The circuitry that actually performs effective-address calculations resides in the IFU, but the ID oversees these operations. The generation of effective addresses is performed within a separate carry look-ahead adder, used with hardware shift logic. The ability to calculate effective addresses independently from instruction execution allows address calculation to be overlapped with computation. The time required to calculate an effective address ranges from zero to four cycles; but, for the most commonly used addressing modes, this time is less than two cycles.

Instructions that require effective addresses are executed by either the ID or the MMU and BCL, thus preserving the pipeline and eliminating delays or resource constraints on the IEU or FPU.

### Micro-Instruction Sequencer and ROM

The MIS is a multipurpose unit designed to help in the execution of instructions that use microcode. All of the processor's microcode is stored in ROM, which is accessed through the MIS. When the ID receives a complex instruction (one that requires microcode to be executed), the MIS supplies the microcode to the IEU as described earlier in the discussion of complex instructions.

The MIS also supplies microcode for floating-point instructions; the power-up and self-test performed during processor initialization; interrupt handling; and fault handling.

The MIS is able to access parts of the processor that are not accessible to a program, such as the cached local register sets and parts of system data structures that have been cached on the chip. This capability offers two benefits. First it allows certain operations such as flushing the local registers sets to be carried out, even though software does not have direct access to these registers. Second, it enables the processor to execute complex process management operations very quickly.

### Instruction Execution Unit

The IEU contains the Arithmetic Logic Unit (ALU) and the mechanism for register and condition-code scoreboarding. It also manages the 16 global registers and the 4 sets of 16 local registers.

The ALU performs the following functions for the IEU:

- Addition and subtraction of integers and ordinals
- Moves between registers
- Logical operations
- Bit operations
- Shifts and rotates
- Comparisons

It is capable of performing any of these operations in a single clock cycle.

The IEU can also work with integer literals in the range of 0 to +31, which are encoded in the REG instruction format. This method of encoding literals performs two functions. First, it provides a more compact instruction stream. Second, when a literal is used as an argument for an instruction, the IEU is able to execute the instruction in one less clock cycle.

The IEU handles the reading and writing of global and local registers. It also handles the allocation of local registers sets on procedure calls. The IEU allocates a new set of local registers on each procedure call. If all four register sets become allocated, the IEU automatically flushes the oldest frame to the stack on the next procedure call. The IEU also automatically retrieves any local register frame from the stack when required by a return operation. The majority of procedure calls or returns do not require the processor to flush local registers to memory. Call instructions that can be executed without flushing a register set require only 9 cycles to complete, with the corresponding return taking only 7 cycles.



The register scoreboard provides scoreboarding for the global and local registers. When one or more registers are being used in an operation, they are marked as in use. The register scoreboarding mechanism allows the processor to continue executing subsequent instructions, as long as those instructions do not require the contents of the scoreboarded registers.

A typical event that would cause scoreboarding is a load operation. For a load from memory, the contents of the affected registers are not valid until the MMU and BCL fetch the data and the registers are loaded. For example, consider the sequence:

```
ld    (g1), g0
addi  g2, g3, g4
addi  g5, g4, g6
subi  g0, g6, g6
```

Here, when the MMU and BCL initiate the **ld** operation, register g0 is scoreboarded. As long as subsequent instructions do not require the contents of g0, the ID continues to dispatch instructions. For example, the two **addi** instructions above are executed while the BCL is fetching the data for g0. If g0 is not loaded by the time the **subi** instruction is ready to be executed, the IEU delays execution of the instruction until the loading of g0 has been completed.

If an operation accesses a single register, only that register is scoreboarded. However, if multiple registers are accessed (such as, with the **ldl**, **ldt**, or **ldq** instructions), registers are scoreboarded as shown in Table C-1, according to the base register of the group being accessed.

**Table C-1: Registers Scoreboarded According to Registers Referenced**

Base Register Accessed	Block of Registers Scoreboarded
g0	0-3
g2	0-7
g4	4-7
g6	0-15
g8	8-11
g10	8-15
g12	12-15
g14	0-15

### Register Bypassing

The execution times of instructions in the IEU are dependent on the instruction flow. One feature of the IEU that can enhance processor performance is register bypassing. Register bypassing is a mechanism that allows an instruction that would ordinarily require source operands to be placed in registers to be executed without accessing one or both of the source registers. Register bypassing occurs in either of two circumstances. First, when the IEU executes an instruction with two source operands, register bypassing occurs if one or both of

the operands are literals. Second, register bypassing will also occur when the second of two source operands is the result of the previous instruction. The net result of register bypassing is the saving of one clock cycle. Most instructions that the IEU executes can be executed in a single cycle when register bypassing occurs.

### Floating Point Unit

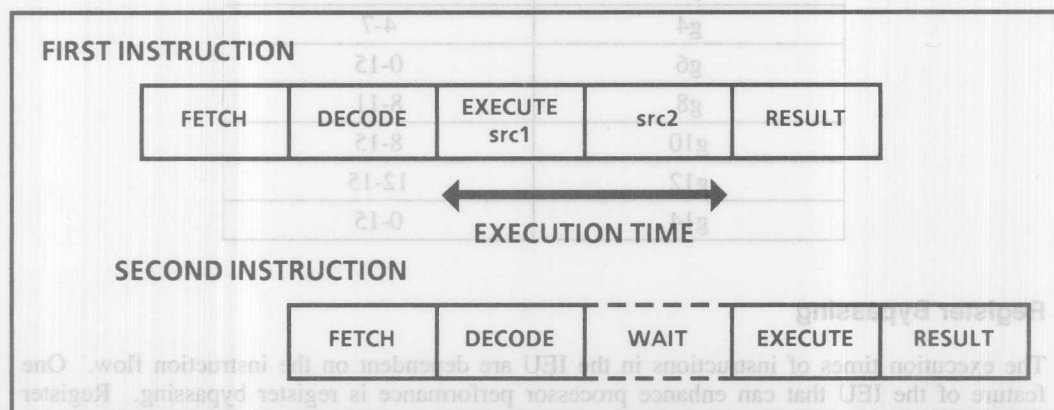
The FPU performs all the floating-point computations for the processor, as well as the integer multiply and divide operations. It shares the resources of the processor. For example, it can use the global and local registers as operands for floating-point operations, and it gets microcode for the execution of complex floating-point instructions from the MIS. It also manages the four 80-bit floating-point registers, which it uses for extended-precision, floating-point calculations.

To perform integer multiplication and several floating-point calculations, the FPU contains a 32-bit integer Booth-Multiplier. This multiplier performs integer multiplication operation in a variable amount of time, depending on the number of significant bits. It is used for integer multiplications and several floating-point calculations.

### EXECUTION TIMES FOR THE 80960 ARCHITECTURE INSTRUCTIONS

This section describes the execution times for the instructions defined the 80960 architecture. As illustrated earlier in this appendix, the execution time for an instruction can vary, according to (1) the types of arguments used and the state of the on-chip resources and (2) how the processor's pipelining and instruction-overlapping features are used.

In the following discussion, an instruction's execution time is defined as the time between the beginning of execution of a decoded instruction and the beginning of execution for the next decoded instruction. For example, the illustration in Figure C-2 shows the execution time of a two operand instruction to be two clocks, with respect to the next instruction to be executed.



**Figure C-2: Execution Time of an Instruction**



## Logical Instructions

The timing of the logical instructions depends on the IEU bypass mechanism described earlier in this appendix, in particular for any instruction of the form:

alu\_instruction *src1*, *src2*, *dst*

If *src1* or *src2* is a literal or if *src2* is the result of the previous operation, a bypass hit occurs. Otherwise, there is no bypass hit and the instruction requires an extra clock to load the second operand. Table C-2 shows the timing of the logical instructions depending on whether or not a bypass hit occurs.

In all the following tables, execution time is given in number of clock cycles.

**Table C-2: Logical Instruction Timing**

Instruction	Normal Case Execution Time (Bypass Hit)	Worst Case Execution Time (Bypass Miss)
and	1	2
nand	1	2
or	1	2
nor	1	2
xor	1	2
xnor	1	2
andnot	1	2
notand	1	2
not	1	1
notor	1	2
ornot	1	2
rotate	1	2
shlo	1	2
shro	1	2
shli	2	3
shri	2	3
shrdi	2	3

## Bit Instructions

The execution times for the bit instructions are also dependent on whether or not a register bypass has occurred or not, as is shown in Table C-3.

**Table C-3: Bit Instruction Timing**

Instruction	Normal Case Execution Time (Bypass Hit)	Worst Case Execution Time (Bypass Miss)
notbit	2	3
setbit	2	3
clrbt	2	3
alterbit	2	3
chkbit	2	3
extract	7	7
modify	8	8

The execution times of the **scanbit** and **spanbit** instructions (shown in Table C-4) depend on condition code scoreboarding. If the condition code is not set by the previous instruction execution, the instruction will complete in one less clock cycle. Execution time is also dependent on the number of bits operated upon.

**Table C-4: Scan and Span Bit Instruction Timing**

Instruction	Best Case Execution Time	Normal Case Execution Time	Worst Case Execution Time
scanbit	8	11	14
spanbit	8	11	14

## Register Moves

The timing of instructions that move data between registers is directly related to the number of words moved. One clock cycle is required to move one (as shown in Table C-5).

**Table C-5: Move Instruction Timing**

Instruction	Execution Time
mov	1
movl	2
movt	3
movq	4

## Integer and Ordinal Arithmetic

The execution times for the basic add and subtract instructions (as shown in Table C-6) depend on register bypass. The normal-case results are achieved when a register bypass occurs.

**Table C-6: Integer and Ordinal Arithmetic Instruction Timing**

Instruction	Normal Case Execution Time (Bypass Hit)	Worst Case Execution Time (Bypass Miss)
addo	1	2
addi	1	2
subo	1	2
subi	1	2
addc	1	2
subc	1	2

Table C-7 shows the execution times of the compare instructions, which also depend on whether or not a bypass hit occurs.

**Table C-7: Compare Instruction Timing**

Instruction	Normal Case Execution Time (Bypass Hit)	Worst Case Execution Time (Bypass Miss)
cmpo	1	2
cmpi	1	2
cmpinco	2	3
cmpdeco	2	3
cmpinci	2	3
cmpdeci	2	3
condmpo	1	2
concmpi	1	2

## Multiply and Divide Instructions

Table C-8 shows the typical instruction execution times for the multiply and divide instructions:

**Table C-8: Multiply and Divide Instruction Timing**

Instruction	Range of Significant Bits	Typical Case Execution Time
<b>mulo</b>	9 to 21	18
<b>muli</b>	9 to 21	18
<b>divi</b>	37	37
<b>divo</b>	37	37
<b>remo</b>	37	37
<b>remi</b>	37	37
<b>modi</b>	37	37
<b>emul</b>	37	24
<b>ediv</b>	37	40

Since the processor contains a Booth Multiplier with early out, the execution times on the multiply and divide instructions (shown in Table C-8) depend on the number of significant bits in the *src1* operand. For example, Table C-9 shows the execution times based on the number of significant bits in *src1*:

**Table C-9: Multiply/Divide Execution Times Based on Significant Bits**

Src1 Significant Bits	Execution Time
2	9
4	10
8	11
32	21

Note that the shift instructions or the add and subtract instructions may be faster than the multiply instructions in certain instances (for example, when multiplying by 3, 5, 15, etc.).

## Branching

Branch instructions are executed directly by the ID and do not require IEU or FPU resources. Because of this, branch instructions can in most cases be programmed so that their execution is overlapped with other operations. Table C-10 lists the ranges of times for execution of branch instructions, from best (maximum overlap) to worst (no overlap). (The instructions in capital letters indicate groups of instructions that branch on condition codes, such as the **BRANCH IF** instructions, **be**, **bg**, **bl**, etc.)

Table C-10: Branch Instruction Timing

Instruction	Best Case Execution Time (CC Available)	Worst Case Execution Time (CC Not Available)
<b>b</b> <sup>1</sup>	0 to 2 (0 to 2)	
BRANCH IF	0 to 2 (0 to 1)	0 to 3 (0 to 2)
<b>bx</b> <sup>1</sup>	0 to 6 (0 to 6)	
BRANCH AND LINK <sup>1</sup>	2 to 8 (2 to 8)	
COMPARE AND BRANCH <sup>2</sup>		4 to 5 (3 to 4)
<b>bbs, bbc</b> <sup>2</sup>		4 to 5 (3 to 4)
TEST IF	0 to 3 (0 to 2)	0 to 4 (0 to 3)
FAULT IF	0 to 2 (0 to 1)	0 to 3 (0 to 2)

## Notes:

1. Condition code is not used.
2. Condition code is set and checked as part of instruction execution.

The second column of numbers lists execution-time ranges for conditional branches in which the condition code was not set in the previous instruction, and the third column lists ranges for branches in which the condition code was set by the previous instruction. Also, the first range in each column is for the case in which the branch is taken, and the range in parentheses is for the case in which the branch is not taken.

When writing optimized code for the 80960MC processor, it is best to perform conditional tests at least two instructions before a conditional branch. This practice allows the execution times in column two to be achieved. It is also important to note that the "not taken" branch case executes in one less cycle, because there is no break in the pipeline. (Remember, instruction time is defined as the time from the start of execution of one instruction to the start of execution of the next instruction. If the pipeline is stalled, the fetch of the next instruction will be delayed one clock. This delay may or may not be hidden by the parallelism of the 80960MC processor).

### Call/Return Instructions

As described earlier in this appendix, the 80960MC processor provides four sets of local registers. When a call instruction is executed, the processor allocates a new set of local registers to the called procedure or interrupt routine. If, when a **call** or **callx** instruction is executed, a set of local registers is available, the processor executes the instruction in 9 clock cycles.

If a set of local registers is not available, the processor flushes the oldest set of registers to the stack in memory to free up a register set. Flushing a set of local registers requires four quad-word stores to memory. Assuming zero-wait-state memory, this operation adds 24 clocks to the 9 clocks normally required to execute a call.

The **ret** (return) instruction normally requires 7 clock cycles. If the local registers being returned to have been flushed to the stack, an additional 24 clocks must be added to this execution time (with zero-wait-state memory) for the processor to reload the local registers from the stack. It is important to note that the processor only reloads the local registers when they are required, thus eliminating unnecessary memory cycles.

### Miscellaneous Complex Instructions

The miscellaneous complex instructions shown in Table C-11 are carried out by the MIS. Their execution times depend on the execution state of the environment at the time of execution. The execution times given here are typical values.

**Table C-11: Miscellaneous Complex Instruction Timing**

Instruction	Execution Time
<b>atadd</b>	17
<b>atmod</b>	20
<b>flushreg</b>	27
<b>mark</b>	6 (not taken)
<b>fmark</b>	6 (plus fault time)
<b>modac</b>	10
<b>modpc</b>	29
<b>modtc</b>	18
<b>lda</b>	1 to 5 (typical 2)
<b>ldphy</b>	17
<b>inspacc</b>	29

### Load Instructions

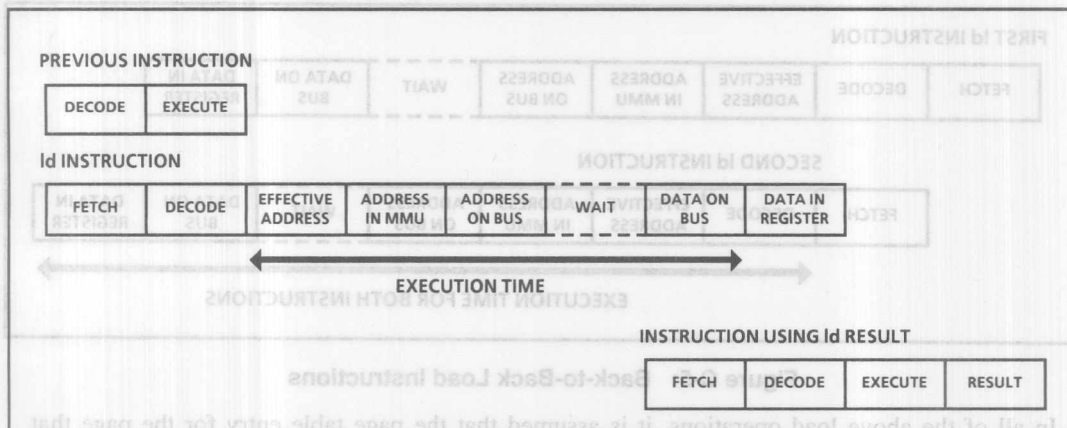
A load instruction requires the following steps:

1. Instruction Fetch
2. Decode
3. Compute Effective Address/Scoreboard Register(s)
4. Address translation through the MMU
5. Place Address on Bus
6. Wait State(s)
7. Receive Data on Bus
8. Place Data in target register

Of these steps, only steps 3 through 8 are included in the definition of execution time for an instruction. The following figures show several examples of load instruction timing depending on where the load instruction is placed in the instruction stream.

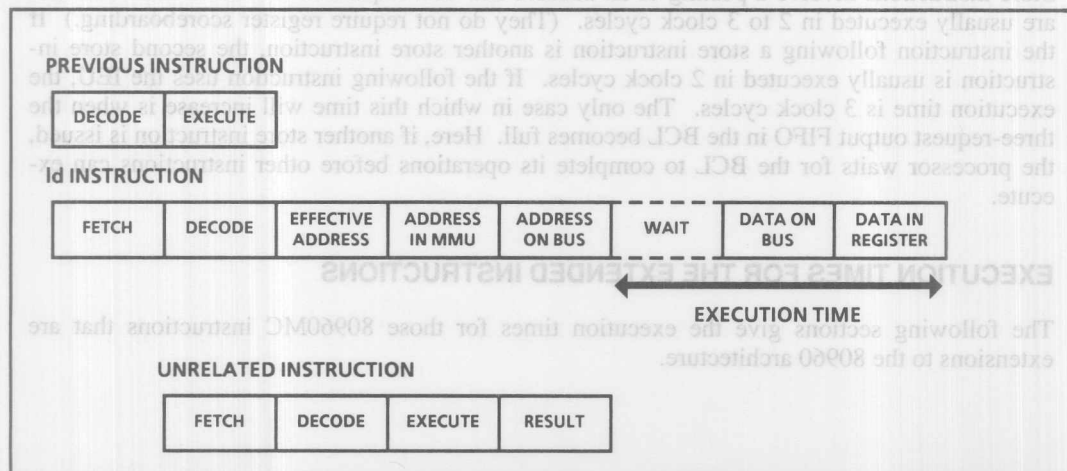


The example in Figure C-3 illustrates a load instruction where the instruction that follows requires the fetched data. Here, the pipeline is stalled while the processor waits for the load to complete. Assuming a one-clock-cycle effective-address calculation, the load will require 4 or 5 clock cycles to be executed, depending on whether or not zero-wait-state memory is used.



**Figure C-3: Load Where the Next Instruction Requires the Fetched Data**

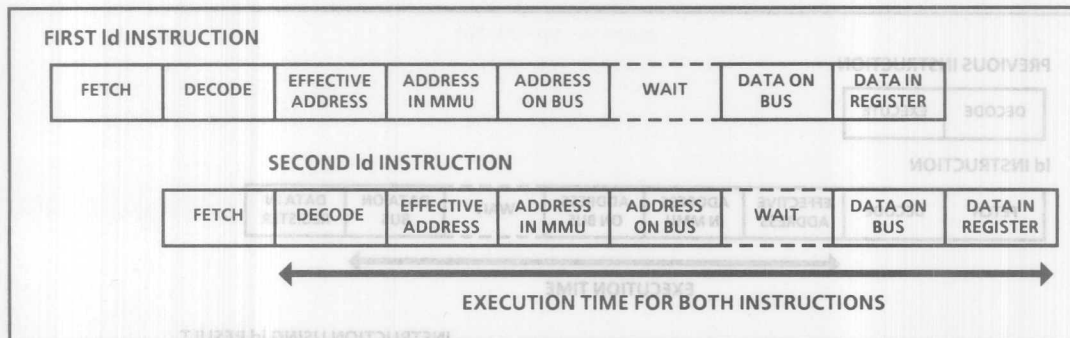
Figure C-4 gives an example of a load instruction where the instruction that follows does not require the data being fetched from memory. Here, the unrelated instruction can be executed while the load is being completed. The 2 clock cycles required to execute the unrelated instruction are then overlapped with the 4 or 5 cycles required to execute the load (again depending on whether or not zero-wait-state memory is used). The load instruction thus requires a net of 1 or 2 clock cycles from the pipeline to be executed.



**Figure C-4: Load Where the Next Instruction Does Not Require the Fetched Data**

Finally, Figure C-5 shows an example of two load instructions being executed back-to-back. These two instructions can be executed in 5 or 6 clock cycles, as long as the number of BCL requests is limited to 3 or less (which is the size of the output request FIFO in the BCL's

control queue). Here, the second load is almost completely overlapped by the first load. Times for multiple word loads will be lengthened 1 cycle plus wait states for each additional word. If more than 3 requests become outstanding, the processor will wait until the number of outstanding load operations goes below the size of the output FIFO.



**Figure C-5: Back-to-Back Load Instructions**

In all of the above load operations, it is assumed that the page table entry for the page that contains the word is present in the TLB (which is normally the case). If not, the translation takes considerably longer, since the processor has to perform several memory reads to thread its way through the segment table and page tables to find the physical address of the page that contains the word to be loaded.

### Store Operations

Store instructions involve a posting of an address and data request to the MMU and BCL and are usually executed in 2 to 3 clock cycles. (They do not require register scoreboarding.) If the instruction following a store instruction is another store instruction, the second store instruction is usually executed in 2 clock cycles. If the following instruction uses the IEU, the execution time is 3 clock cycles. The only case in which this time will increase is when the three-request output FIFO in the BCL becomes full. Here, if another store instruction is issued, the processor waits for the BCL to complete its operations before other instructions can execute.

### EXECUTION TIMES FOR THE EXTENDED INSTRUCTIONS

The following sections give the execution times for those 80960MC instructions that are extensions to the 80960 architecture.

## Decimal Instructions

Table C-12 shows the instruction times for the decimal instructions.

**Table C-12: Decimal Instruction Timing**

Instruction	Execution Time
<b>dmovt</b>	7
<b>daddc</b>	8
<b>dsubc</b>	8

## Floating-Point Instructions

Table C-13 shows the instruction execution times for the simple floating-point instructions. Where applicable, a range and a typical observed average are given.

The instructions given in Table C-14 consist of the complex floating point instructions. Only typical instruction execution rates are given here. In many cases, the clock count can vary by 30-40%. Execution time is dependent on the operands.

It is important to note that the complex floating-point instructions are interruptible. When an interrupt is received while one of these instructions is being executed, the processor can suspend execution, service the external request, then resume execution of the instruction.

## Process-Management Instructions

The MIS executes the process management instructions. The execution times for these instructions depend heavily on the state of the execution environment when execution of the instruction begins. For example, if a **signal** instruction is executed, the execution time will vary depending on whether or not there is a process waiting at the semaphore.

Table C-15 gives typical execution times for these instructions. The following assumptions are made in computing these times:

- The system is assumed to be a single-processor system
- Regions are assumed to be paged
- Faults do not occur
- When enqueueing occurs, the queue is empty
- When dequeuing occurs, one entry is on the queue
- All communication ports are assumed to be FIFO ports
- Process preemption does not occur as the result of any operation

Table C-13: Simple Floating-Point Instruction Timing

Instruction	Execution Time
movr	5
movrl	5 to 7
movre	7 to 8
cpysre	8
cpysre	8
addr	9 to 17 (typical 10)
addrl	12 to 20 (typical 13)
subr	9 to 17 (typical 10)
subrl	12 to 20 (typical 13)
mulr	11 to 22 (typical 20)
mulrl	14 to 43 (typical 36)
divr	35
divrl	77
cmpr	10
cmprl	12
cmpor	10
cmporl	12
cvtri	25 to 33
cvtril	26 to 35
cvtilr	41 to 45
cvtilr	42 to 46
cvtzri	41 to 45
cvtzril	42 to 46
roundr	56 to 69
roundrl	56 to 70
scaler	28
scalerl	30
logbnr	32 to 41
logbnrl	32 to 43
classr	22 to 24
classrl	22 to 24

Table C-14: Complex Floating-Point Instruction Timing

Instruction	Execution Time
<b>sqrtrl</b>	104
<b>expr</b>	300
<b>exprl</b>	334
<b>logepr</b>	400
<b>logeprl</b>	420
<b>logr</b>	438
<b>logrl</b>	438
<b>remr</b>	(67 to 75878)
<b>remrl</b>	(67 to 75878)
<b>atanr</b>	267
<b>atanrl</b>	350
<b>cosr</b>	406
<b>cosrl</b>	441
<b>tanr</b>	293
<b>tanrl</b>	323

Table C-15: Process-Management Instruction Timing

Instruction	Execution Time
<b>wait</b>	47 (no blocking)
<b>condwait</b>	47
<b>signal</b>	42 (no waiting process) 137 (waiting process)
<b>send</b>	110 (no waiting process) 172 (waiting process)
<b>receive</b>	73 (message available)
<b>condrec</b>	69 (message not available) 92 (message available)
<b>schedprcs</b>	107
<b>sendserv</b>	185
<b>ldtime</b>	15
<b>saveprcs</b>	200
<b>resumprcs</b>	375

Table C-14: Complex Floating-Point Instruction Timing

Instruction	Execution Time
tanhl	323
tanh	293
costl	441
cost	406
atanhl	320
atanr	267
remhl	(67 to 72878)
remr	(67 to 72878)
logrl	438
logr	438
logcprl	420
logcpr	400
expri	334
exp	300
sqrtrl	104

Table C-15: Process-Management Instruction Timing

Instruction	Execution Time
wait	47 (no blocking)
condwait	47
signal	42 (no waiting process) 137 (waiting process)
send	110 (no waiting process) 172 (waiting process)
receive	73 (message available)
condrec	69 (message not available) 92 (message available)
schedprcs	107
sendserv	182
ldtime	12
saveprcs	200
resumpres	372



---

## *Appendix D*

### *Initialization Code*

---

---

## Appendix D Initialization Code

---

## APPENDIX D INITIALIZATION CODE

This appendix provides an example of the initialization code required to initialize the 80960MC processor.

### OVERVIEW

The code given in this appendix demonstrates one of the methods that can be used to initialize the 80960MC processor. To use this code, the programmer must assemble (and compile, in the case of the C program modules) the individual files into object modules. These modules must then be loaded into ROM (generally EPROM). The resulting EPROM will contain the following:

- An initial memory image (as shown in Figure 9-5)
- An interrupt table
- A fault table
- A system procedure table
- A set of dummy interrupt and fault handler routines
- A dispatch port
- A set of dummy system procedures
- Two small processes

The dummy interrupt and fault handler routines merely perform a return to the initialization code if an interrupt or fault occurs during initialization. Likewise, the dummy system procedures perform returns. These routines may be changed to suit the needs of a particular application.

Each process consists of a PCB and a code block. The code block is located in physical memory; however, one of the jobs of the initialization code will be to map the code into a virtual memory page.

The dispatch port has the two processes already queued to it.

When the processor's RESET pin is asserted, the processor performs its self test and comes up in physical mode. The processor then begins executing the initialization code. This code directs the processor to perform the following rudimentary steps of initialization:

1. Copy the PRCB from the IMI into RAM.
2. Copy the interrupt table into RAM.
3. Copy the dispatch port in RAM.
4. Copy page tables for the two processes in RAM.

5. Copy a page table for a region 3 in RAM, to be shared by the two processes.
6. Copy the PCBs for the two processes into RAM.
7. Execute a restart processor IAC, to enable the processor to load the new pointers to PRCB and interrupt table. During restart, the processor is brought up in virtual mode and in the idle state.

The PRCB, interrupt table, dispatch port, and process PCBs are copied into RAM because these data structures have fields that the processor must be able to write.

In a system where processes are created dynamically, the segment table would also have to be copied into RAM during initialization. In this example, the segment table remains in ROM. The pointers in the segment table to the page tables, PCBs, dispatch port, and system procedure table are predefined to point to the locations in RAM where these data structures are to be loaded during initialization.

Prior to restarting the processor, additional initialization steps can be carried out to configure the processor for a particular application. The following items are examples of further initialization actions that might be included in the initialization code:

- Copy the segment table into RAM (as discussed above).
- Copy new interrupt handler routines into RAM and change the pointers in the interrupt table to point to these new routines.
- Copy the fault table into RAM; copy new fault handler routines into RAM; change the pointers in the fault table to point to the new fault handler routines; and change the pointer in the PRCB to point to the relocated fault table.
- Create a new system procedure table in RAM; copy the system procedures into RAM; change the pointer in the PRCB to point to the new system procedure table.
- Create additional processes, made up of page tables for the process address space, a PCB, and code and data for the process.

Alternatively, the interrupt handler routines, fault handler routines, and system procedures can all be loaded into ROM.

Following the restart of the processor, the processor checks the dispatch port. It dispatches the first process and begins executing it. It executes the process for one time slice of 4096 ticks, then dispatches the second process. It continues to switch back and forth between the two processes in this manner.

#### EXAMPLE CODE

The example code consists of the following sixteen files:

- startup.s
- f\_table.lst
- i\_table.lst

- initial\_frame.lst
- macs.m4
- f\_handle.c
- i\_handle.c
- fix\_pte.c
- prog1.c
- prog2.c
- led.h
- pass1.ld
- pass1a.ld
- pass2.ld

The *startup.s*, *f\_table.lst*, *i\_table.lst*, *initial\_frame.lst*, and *macs.m4* files contain assembly code for the Intel 80960MC Assembler. (The files with an *.lst* extensions are listings from the assembler that include assembly code, such as would be included in an *.s* file, and the resulting object code. The *macs.m4* file contains assembler code for macros.) The code in these files is used to build the initial memory image. The *startup.s* code builds all of the system data structures except the interrupt table and fault table, which are built by the *i\_table.lst* and *f\_table.lst* code, respectively. The *startup.s* code uses the macros in *macs.m4*. Also, the *startup.s* code contains the initialization code that the processor executes following the first stage of initialization. The *initial\_frame.lst* code creates a stack frame for each process.

*f\_handle.c*, *i\_handle.c*, and *fix\_pte.c* files contain C program modules that are also used to build the initial memory image. The *f\_handle.c* and *i\_handle.c* programs create the dummy fault and interrupt handler routines; the *fix\_pte.c* program creates the page tables.

The *prog1.c*, *prog2.c*, and *led.h* files contain C program modules for the two processes.

Finally, the *pass1.ld*, *pass1a.ld*, and *pass2.ld* files contain instructions for the loader.

The following steps describe how to use the code in these files:

1. Assemble the assembly code in files *startup.s*, *f\_table.lst*, *i\_table.lst*, *initial\_frame.lst*, and *macs.m4*. (Here the ".s" files are made up of the assembly code only from the ".lst" files listed above.)
2. Compile the C code in files *f\_handle.c*, *i\_handle.c*, *fix\_pte.c*, *prog1.c*, and *prog2.c*. The *led.h* code is included in the *prog1.c* and *prog2.c* code.
3. Run the *pass1.ld* command file. The script in this file do two things. First, it links the object modules *prog1.o* and *initial\_frame.o*, using the 80960 linker. This operation creates the virtual address space for process 1, with code starting at address  $0_{16}$ , data at address  $40000000_{16}$ , the stack at address  $80000000_{16}$ , and region 3 at  $C0000000_{16}$ . Second, the interrupt and fault tables are located in region 3. (The interrupt and faults tables are not related to process 1. They are located using *pass1.ld*, merely for convenience.)

4. Run the *pass1.ld* command file. The script in this file create a virtual address space for process 2, by linking the object modules *prog2.o* and *initial\_frame.0*.
5. Run the *pass2.ld* command file. The script in this file combine the two processes with the initial memory image. The script in *pass2.ld* directs the linker to locate the linked code at address 0.
6. Burn the output file from *pass2.ld* from the linker in an EPROM.

### startup.s

```

/*----- externals -----*/
.globl seg_table_ptr
.globl prcb_ptr
.globl start_ip
.globl cs1
.globl _p1_region_0_pte
.globl _p1_region_1_pte
.globl _p1_region_2_pte
.globl _p2_region_0_pte
.globl _p2_region_1_pte
.globl _p2_region_2_pte
.globl _region_3_pte

/*----- Core initialization block -----*/

.word seg_table_ptr
.word prcb_ptr
.word 0
.word start_ip
.word cs1 /* calculated at link time */
.word 0 /* cs1 = --(segtab + PRCB + startup) */
.word 0
.word -1

/*----- segment table offsets -----*/

.set sys_proc_table_st,2
.set p1_region_0_st,3
.set p1_region_1_st,4
.set p1_region_2_st,5
.set region_3_st,6
.set d_port_st,7
.set segtab_st,8
.set p2_region_0_st,9
.set p2_region_1_st,10
.set p2_region_2_st,11
.set LPCB1_st,12
.set LPCB2_st,13

/*----- region sizes -----*/
/* nominal object size = (size+1) * 64kb */

```



```

.set    p1_region_0_size,0
.set    p1_region_1_size,0
.set    p1_region_2_size,0
.set    p2_region_0_size,0
.set    p2_region_1_size,0
.set    p2_region_2_size,0
.set    region_3_size,0x3f /* as large as possible */

/* ----- initial PRCB ----- */
/*
This PRCB (Processor Control Block) is used to bring
the 80960 out of reset and into an executing state. The
processor will set up all necessary tables and structures,
then restart itself using the Linear PRCB (below)

*/

.align 6
prcb_ptr:
.word 0x0 /* 0 - reserved */
.word 0x00000008 /* 4 - processor state = idle */
.word 0x0 /* 8 - reserved */
.word 0x0 /* 12 - current process */
.word 0x0 /* 16 - dispatch port */
.word intr_table /* 20 - table physical address */
.word _intr_stack /* 24 - interrupt stack pointer */
/* Note: G15 is the frame pointer and
is initialized to int_stack at reset */
.word 0x0 /* 28 - reserved */
SS(region_3_st) /* 32 - region 3 */
.word sys_proc_table /* 36 - system procedure table */
.word fault_table /* 40 - fault table */
.word 0x0 /* 44 - reserved */
.space 12 /* 48 - reserved */
.word 0x0 /* 60 - reserved */
.space 8 /* 64 - idle time */
.word 0x0 /* 72 - system error fault */
.word 0x0 /* 76 - reserved */
.space 48 /* 80 - resumption record */
.space 44 /* 128 - system error fault record */

/* ----- linear PRCB ----- */
/*
This block is set up at the end of the
period. When the time expires, the
processor should be ready to be
task at the end of the dispatch
*/
.align 12
lprcb_ptr:
.word 0x0 /* 0 - reserved */
.word (1<<10)|(1<<3) /* 4 - addr. trans. on (linear)
, state idle */
.word 0x0 /* 8 - reserved */
.word 0x0 /* 12 - current process */
SS(d_port_st) /* 16 - dispatch port */
.word intr_table /* 20 - table physical address */
.word 0xc0000000 /* 24 - interrupt stack pointer
(beginning of region 3) */
/* Note: G15 is the frame pointer and
is initialized to int_stack at reset */
.word 0x0 /* 28 - reserved */
SS(region_3_st) /* 32 - region 3 */
SS(sys_proc_table_st) /* 36 - system procedure table */
.word fault_table /* 40 - fault table phys. addr. */
.word 0x0 /* 44 - reserved */
.space 12 /* 48 - reserved */
.word 0x0 /* 60 - reserved */
.space 8 /* 64 - idle time */
.word 0x0 /* 72 - system error fault */
.word 0x0 /* 76 - reserved */
.space 48 /* 80 - resumption record */
.space 44 /* 128 - system error fault record */
.text

/* ***** */
/* The system procedure table will only be used if software puts the
/* processor into user mode and makes a supervisor procedure call.
*/

.align 6
sys_proc_table:
.word 0 /* Reserved */

```

D-6

```

.word 0          /* reserved */
.word 0          /* reserved */
.word 0          /* reserved */
.word 0          /* reserved */
SS(p2_region_0_st) /* region 0 Segment selector */
SS(p2_region_1_st) /* region 1 Segment Selector */
SS(p2_region_2_st) /* region 2 Segment Selector */
.word 0x10000000 /* arith. controls:inexact mask */
.word 0          /* reserved */
.word 0x1000     /* next time slice */
.space 8         /* execution time */
.space 48        /* resumption record */
.space 60        /* global registers g0..gl4 */
.word 0x80000000 /* initial frame pointer */
.space 48        /* floating point registers */

/* ----- initial segment table ----- */
.align 12
seg_table_ptr:
    null_seg()          /* ste 0 No entry */
    null_seg()          /* ste 1 No entry */
    small_seg(sys_proc_table) /* ste 2 */
    paged_region(p1_region_0_pte,p1_region_0_size) /* ste 3 */
    paged_region(p1_region_1_pte,p1_region_1_size) /* ste 4 */
    paged_region(p1_region_2_pte,p1_region_2_size) /* ste 5 */
    paged_region(region_3_pte,region_3_size) /* ste 6 */
    port_seg(d_port) /* ste 7 */
    simple_region(seg_table_ptr) /* ste 8 */
    paged_region(p2_region_0_pte,p2_region_0_size) /* ste 9 */
    paged_region(p2_region_1_pte,p2_region_1_size) /* ste 10 */
    paged_region(p2_region_2_pte,p2_region_2_size) /* ste 11 */
    small_seg(lpcb1_ptr) /* ste 12 */
    small_seg(lpcb2_ptr) /* ste 13 */

/* ----- other misc. stuff ----- */
/* these are the entries for the page tables in memory. We will allocate
page tables for regions 0 1 and 2 for each process based on region
size. This value will be provided at linkage time by the linker,
and allow the second pass of the linker to create page tables of
the appropriate size. Region 3 page tables contain entries
for memory mapped I/O (located at physical address 0x11000000,
0x12000000, 0x13000000, 0x14000000) which will be mapped to the
corresponding linear addresses */
.data
.align 6
_p1_region_0_pte: .space (p1_region_0_size+1)*64
_p1_region_1_pte: .space (p1_region_1_size+1)*64
_p1_region_2_pte: .space (p1_region_2_size+1)*64
_p2_region_0_pte: .space (p2_region_0_size+1)*64
_p2_region_1_pte: .space (p2_region_1_size+1)*64
_p2_region_2_pte: .space (p2_region_2_size+1)*64
_region_3_pte: .space (256*3)*4
                page_entry(0x11000000) /* lin: 0xc0300000 */
                page_entry(0x12000000) /* lin: 0xc0301000 */
                page_entry(0x13000000) /* lin: 0xc0302000 */
                page_entry(0x14000000) /* lin: 0xc0303000 */
                .space (256-4)*4

/* the space below contains the dispatch port. This structure will be
statically created in this module, to indicate a priority port
with processes ready to dispatch. The entry for Priority
0 contains a head pointer to process 1 and a tail pointer to
process 2. */
.align 6
d_port: .word (1<<16) /* Priority Port */
        .word 0x1     /* 1 message at 0 */
        SS(LPCB1_st) /* Queue Head prior 0 */
        SS(LPCB2_st) /* Queue Tail prior 0 */
        .space 31*2 /* Head & Tail for 1-31
                    proirity entries */

/* The processor begins code execution here after reset. */
.align 8

```

D-8

## f\_table.lst

tbl.eidasf\_i

```

0 0000          # 1 "f_table.s"  "b.eidasf_i" i #          0000 0
1 0000          00000000 00000000 00000000 00000000 0000 1
2 0000          00000000 00000000 00000000 00000000 0000 2
3 0000          .globl fault_table 0000 3
4 0000          .align 8 0000 4
5 0000          fault_table: 0000 5
6 0000 00000000 00000000 00000000 00000000 0000 6
7 0004 00000000 00000000 00000000 00000000 0000 7
8 0008 00000000 00000000 00000000 00000000 0000 8
9 000c 00000000 00000000 00000000 00000000 0000 9
10 0010 00000000 00000000 00000000 00000000 0000 10
11 0014 00000000 00000000 00000000 00000000 0000 11
12 0018 00000000 00000000 00000000 00000000 0000 12
13 001c 00000000 00000000 00000000 00000000 0000 13
14 0020 00000000 00000000 00000000 00000000 0000 14
15 0024 00000000 00000000 00000000 00000000 0000 15
16 0028 00000000 00000000 00000000 00000000 0000 16
17 002c 00000000 00000000 00000000 00000000 0000 17
18 0030 00000000 00000000 00000000 00000000 0000 18
19 0034 00000000 00000000 00000000 00000000 0000 19
20 0038 00000000 00000000 00000000 00000000 0000 20
21 003c 00000000 00000000 00000000 00000000 0000 21
22 0040 00000000 00000000 00000000 00000000 0000 22
23 0044 00000000 00000000 00000000 00000000 0000 23
24 0048 00000000 00000000 00000000 00000000 0000 24
25 004c 00000000 00000000 00000000 00000000 0000 25
26 0050 00000000 00000000 00000000 00000000 0000 26
27 0054 00000000 00000000 00000000 00000000 0000 27
28 0058 00000000 00000000 00000000 00000000 0000 28
29 005c 00000000 00000000 00000000 00000000 0000 29
30 0060 00000000 00000000 00000000 00000000 0000 30
31 0064 00000000 00000000 00000000 00000000 0000 31
32 0068 00000000 00000000 00000000 00000000 0000 32
33 006c 00000000 00000000 00000000 00000000 0000 33
34 0070 00000000 00000000 00000000 00000000 0000 34
35 0074 00000000 00000000 00000000 00000000 0000 35
36 0078 00000000 00000000 00000000 00000000 0000 36
37 007c 00000000 00000000 00000000 00000000 0000 37
38 0080 00000000 00000000 00000000 00000000 0000 38
39 0084 00000000 00000000 00000000 00000000 0000 39
40 0088 00000000 00000000 00000000 00000000 0000 40
41 008c 00000000 00000000 00000000 00000000 0000 41
42 0090 00000000 00000000 00000000 00000000 0000 42
43 0094 00000000 00000000 00000000 00000000 0000 43
44 0098 00000000 00000000 00000000 00000000 0000 44
45 009c 00000000 00000000 00000000 00000000 0000 45
46 00a0 00000000 00000000 00000000 00000000 0000 46
47 00a4 00000000 00000000 00000000 00000000 0000 47
48 00a8 00000000 00000000 00000000 00000000 0000 48
49 00ac 00000000 00000000 00000000 00000000 0000 49
50 00b0 00000000 00000000 00000000 00000000 0000 50
51 00b4 00000000 00000000 00000000 00000000 0000 51
52 00b8 00000000 00000000 00000000 00000000 0000 52
53 00bc 00000000 00000000 00000000 00000000 0000 53
54 00c0 00000000 00000000 00000000 00000000 0000 54
55 00c4 00000000 00000000 00000000 00000000 0000 55
56 00c8 00000000 00000000 00000000 00000000 0000 56
57 00cc 00000000 00000000 00000000 00000000 0000 57
58 00d0 00000000 00000000 00000000 00000000 0000 58
59 00d4 00000000 00000000 00000000 00000000 0000 59
60 00d8 00000000 00000000 00000000 00000000 0000 60
61 00dc 00000000 00000000 00000000 00000000 0000 61
62 00e0 00000000 00000000 00000000 00000000 0000 62
63 00e4 00000000 00000000 00000000 00000000 0000 63
64 00e8 00000000 00000000 00000000 00000000 0000 64
65 00ec 00000000 00000000 00000000 00000000 0000 65
66 00f0 00000000 00000000 00000000 00000000 0000 66
67 00f4 00000000 00000000 00000000 00000000 0000 67
68 00f8 00000000 00000000 00000000 00000000 0000 68
69 00fc 00000000 00000000 00000000 00000000 0000 69

```

## i\_table.lst

i\_table.s

```

0 0000          # 1 "i_table.s"
1 0000          .globl intr_table
2 0000          .align 6
3 0000          intr_table:
4 0000          .word 0          # Pending Priorities 0
5 0000 00000000 .fill 8,4,0          # pending Interrupts 4 + (0->7)*4
6 0004          .word _user_intrh; # interrupt table entry
7 0024 00000000 .word _user_intrh; # interrupt table entry
8 0028 00000000 .word _user_intrh; # interrupt table entry
9 002c 00000000 .word _user_intrh; # interrupt table entry
10 0030 00000000 .word _user_intrh; # interrupt table entry
11 0034 00000000 .word _user_intrh; # interrupt table entry
12 0038 00000000 .word _user_intrh; # interrupt table entry
13 003c 00000000 .word _user_intrh; # interrupt table entry
14 0040 00000000 .word _user_intrh; # interrupt table entry
15 0044 00000000 .word _user_intrh; # interrupt table entry
16 0048 00000000 .word _user_intrh; # interrupt table entry
17 004c 00000000 .word _user_intrh; # interrupt table entry
18 0050 00000000 .word _user_intrh; # interrupt table entry
19 0054 00000000 .word _user_intrh; # interrupt table entry
20 0058 00000000 .word _user_intrh; # interrupt table entry
21 005c 00000000 .word _user_intrh; # interrupt table entry
22 0060 00000000 .word _user_intrh; # interrupt table entry
23 0064 00000000 .word _user_intrh; # interrupt table entry
24 0068 00000000 .word _user_intrh; # interrupt table entry
25 006c 00000000 .word _user_intrh; # interrupt table entry
26 0070 00000000 .word _user_intrh; # interrupt table entry
27 0074 00000000 .word _user_intrh; # interrupt table entry
28 0078 00000000 .word _user_intrh; # interrupt table entry
29 007c 00000000 .word _user_intrh; # interrupt table entry
30 0080 00000000 .word _user_intrh; # interrupt table entry
31 0084 00000000 .word _user_intrh; # interrupt table entry
32 0088 00000000 .word _user_intrh; # interrupt table entry
33 008c 00000000 .word _user_intrh; # interrupt table entry
34 0090 00000000 .word _user_intrh; # interrupt table entry
35 0094 00000000 .word _user_intrh; # interrupt table entry
36 0098 00000000 .word _user_intrh; # interrupt table entry
37 009c 00000000 .word _user_intrh; # interrupt table entry
38 00a0 00000000 .word _user_intrh; # interrupt table entry
39 00a4 00000000 .word _user_intrh; # interrupt table entry
40 00a8 00000000 .word _user_intrh; # interrupt table entry
41 00ac 00000000 .word _user_intrh; # interrupt table entry
42 00b0 00000000 .word _user_intrh; # interrupt table entry
43 00b4 00000000 .word _user_intrh; # interrupt table entry
44 00b8 00000000 .word _user_intrh; # interrupt table entry
45 00bc 00000000 .word _user_intrh; # interrupt table entry
46 00c0 00000000 .word _user_intrh; # interrupt table entry
47 00c4 00000000 .word _user_intrh; # interrupt table entry
48 00c8 00000000 .word _user_intrh; # interrupt table entry
49 00cc 00000000 .word _user_intrh; # interrupt table entry
50 00d0 00000000 .word _user_intrh; # interrupt table entry
51 00d4 00000000 .word _user_intrh; # interrupt table entry
52 00d8 00000000 .word _user_intrh; # interrupt table entry
53 00dc 00000000 .word _user_intrh; # interrupt table entry
54 00e0 00000000 .word _user_intrh; # interrupt table entry
55 00e4 00000000 .word _user_intrh; # interrupt table entry
56 00e8 00000000 .word _user_intrh; # interrupt table entry
57 00ec 00000000 .word _user_intrh; # interrupt table entry
58 00f0 00000000 .word _user_intrh; # interrupt table entry
59 00f4 00000000 .word _user_intrh; # interrupt table entry
60 00f8 00000000 .word _user_intrh; # interrupt table entry
61 010c 00000000 .word _user_intrh; # interrupt table entry
62 0100 00000000 .word _user_intrh; # interrupt table entry
63 0104 00000000 .word _user_intrh; # interrupt table entry
64 0108 00000000 .word _user_intrh; # interrupt table entry
65 010c 00000000 .word _user_intrh; # interrupt table entry
66 0110 00000000 .word _user_intrh; # interrupt table entry
67 0114 00000000 .word _user_intrh; # interrupt table entry
68 0118 00000000 .word _user_intrh; # interrupt table entry
69 011c 00000000 .word _user_intrh; # interrupt table entry
70 0120 00000000 .word _user_intrh; # interrupt table entry
71 0124 00000000 .word _user_intrh; # interrupt table entry
72 0128 00000000 .word _user_intrh; # interrupt table entry
73 012c 00000000 .word _user_intrh; # interrupt table entry
74

```



D-11



D-12

```

228 0398 00000000 .word _user_intrh; # interrupt table entry 209
229 039c 00000000 .word _user_intrh; # interrupt table entry 210
230 03a0 00000000 .word _user_intrh; # interrupt table entry 211
231 03a4 00000000 .word _user_intrh; # interrupt table entry 212
232 03a8 00000000 .word _user_intrh; # interrupt table entry 213
233 03ac 00000000 .word _user_intrh; # interrupt table entry 214
234 03b0 00000000 .word _user_intrh; # interrupt table entry 215
235 03b4 00000000 .word _user_intrh; # interrupt table entry 216
236 03b8 00000000 .word _user_intrh; # interrupt table entry 217
237 03bc 00000000 .word _user_intrh; # interrupt table entry 218
238 03c0 00000000 .word _user_intrh; # interrupt table entry 219
239 03c4 00000000 .word _user_intrh; # interrupt table entry 220
240 03c8 00000000 .word _user_intrh; # interrupt table entry 221
241 03cc 00000000 .word _user_intrh; # interrupt table entry 222
242 03d0 00000000 .word _user_intrh; # interrupt table entry 223
243 03d4 00000000 .word _user_intrh; # interrupt table entry 224
244 03d8 00000000 .word _user_intrh; # interrupt table entry 225
245 03dc 00000000 .word _user_intrh; # interrupt table entry 226
246 03e0 00000000 .word _user_intrh; # interrupt table entry 227
247 03e4 00000000 .word _user_intrh; # interrupt table entry 228
248 03e8 00000000 .word _user_intrh; # interrupt table entry 229
249 03ec 00000000 .word _user_intrh; # interrupt table entry 230
250 03f0 00000000 .word _user_intrh; # interrupt table entry 231
251 03f4 00000000 .word _user_intrh; # interrupt table entry 232
252 03f8 00000000 .word _user_intrh; # interrupt table entry 233
253 03fc 00000000 .word _user_intrh; # interrupt table entry 234
254 0400 00000000 .word _user_intrh; # interrupt table entry 235
255 0404 00000000 .word _user_intrh; # interrupt table entry 236
256 0408 00000000 .word _user_intrh; # interrupt table entry 237
257 040c 00000000 .word _user_intrh; # interrupt table entry 238
258 0410 00000000 .word _user_intrh; # interrupt table entry 239
259 0414 00000000 .word _user_intrh; # interrupt table entry 240
260 0418 00000000 .word _user_intrh; # interrupt table entry 241
261 041c 00000000 .word _user_intrh; # interrupt table entry 242
262 0420 00000000 .word _user_intrh; # interrupt table entry 243
263 0424 00000000 .word _user_intrh; # interrupt table entry 244
264 0428 00000000 .word _user_intrh; # interrupt table entry 245
265 042c 00000000 .word _user_intrh; # interrupt table entry 246
266 0430 00000000 .word _user_intrh; # interrupt table entry 247
267 0434 00000000 .word _user_intrh; # interrupt table entry 248
268 0438 00000000 .word _user_intrh; # interrupt table entry 249
269 043c 00000000 .word _user_intrh; # interrupt table entry 250
270 0440 00000000 .word _user_intrh; # interrupt table entry 251
271 0444 00000000 .word _user_intrh; # interrupt table entry 252
272 0448 00000000 .word _user_intrh; # interrupt table entry 253
273 044c 00000000 .word _user_intrh; # interrupt table entry 254
274 0450 00000000 .word _user_intrh; # interrupt table entry 255

```





## f\_handle.c

```

user_override()      {}
user_trace()         {}
user_arithmetic()    {}
user_real_arithmetic() {}
user_constraint()    {}
user_vm()            {}
user_protection()    {}
user_structural()    {}
user_type()          {}
user_reserved_11f()  {}
user_process()       {}
user_descriptor()    {}
user_event()         {}
user_reserved()      {}
user_operation()     {}
user_machine()       {}

```

## i\_handle.c

```

user_intrh()
{
}

```

## fix\_pte.c

```

/*
fix_ptes()

This module "fills in" the appropriate page table entries
with the physical address (obtained at link time) and the
page table attributes. These tables are built by the
processor before the processor goes into "linear mode"
*/
extern unsigned long p1_reg_0_PA;
extern unsigned long p1_reg_1_PA;
extern unsigned long p1_reg_2_PA;
extern unsigned long reg_3_PA;
extern unsigned long p1_reg_0_len;
extern unsigned long p1_reg_1_len;
extern unsigned long p1_reg_2_len;
extern unsigned long reg_3_len;
extern unsigned long p1_region_0_pte;
extern unsigned long p1_region_1_pte;
extern unsigned long p1_region_2_pte;
extern unsigned long region_3_pte;
extern unsigned long p2_reg_0_PA;
extern unsigned long p2_reg_1_PA;
extern unsigned long p2_reg_2_PA;
extern unsigned long p2_reg_0_len;
extern unsigned long p2_reg_1_len;
extern unsigned long p2_reg_2_len;
extern unsigned long p2_region_0_pte;
extern unsigned long p2_region_1_pte;
extern unsigned long p2_region_2_pte;

fix_ptes()
{
    unsigned long i, *pte_ptr, pa_addr;

    /* build page table entries for region 0 */
    pa_addr = (unsigned long) &p1_reg_0_PA;
    pte_ptr = &p1_region_0_pte;
    for (i=0; i< (unsigned long) &p1_reg_0_len; i+=0x1000) {
        *pte_ptr++ = pa_addr | 0xC7; /* present, user/supervisor r/w */
        pa_addr += 0x1000;
    }
    pa_addr = (unsigned long) &p2_reg_0_PA;
    pte_ptr = &p2_region_0_pte;

```

```

for (i=0;i< (unsigned long) &p2_reg_0_len;i+=0x1000) {
    *pte_ptr++ = pa_addr | 0xC7; /* present, user/supervisor r/w */
    pa_addr += 0x1000;
}

/* build page table entries for region 1 */
pa_addr = (unsigned long) &p1_reg_1_PA;
pte_ptr = &p1_region_1_pte;
for (i=0;i< (unsigned long) &p1_reg_1_len;i+=0x1000) {
    *pte_ptr++ = pa_addr | 0xC7; /* present, user/supervisor r/w */
    pa_addr += 0x1000;
}

pa_addr = (unsigned long) &p2_reg_1_PA;
pte_ptr = &p2_region_1_pte;
for (i=0;i< (unsigned long) &p2_reg_1_len;i+=0x1000) {
    *pte_ptr++ = pa_addr | 0xC7; /* present, user/supervisor r/w */
    pa_addr += 0x1000;
}

/* build page table entries for region 2 */
pa_addr = (unsigned long) &p1_reg_2_PA;
pte_ptr = &p1_region_2_pte;
for (i=0;i< (unsigned long) &p1_reg_2_len;i+=0x1000) {
    *pte_ptr++ = pa_addr | 0xC7; /* present, user/supervisor r/w */
    pa_addr += 0x1000;
}

pa_addr = (unsigned long) &p2_reg_2_PA;
pte_ptr = &p2_region_2_pte;
for (i=0;i< (unsigned long) &p2_reg_2_len;i+=0x1000) {
    *pte_ptr++ = pa_addr | 0xC7; /* present, user/supervisor r/w */
    pa_addr += 0x1000;
}

/* build page table entries for region 3 */
pa_addr = (unsigned long) &reg_3_PA;
pte_ptr = &region_3_pte;
for (i=0;i< (unsigned long) &reg_3_len;i+=0x1000) {
    *pte_ptr++ = pa_addr | 0xC7; /* present, user/supervisor r/w */
    pa_addr += 0x1000;
}
}

```

# prog1.c

```
#include "vled.h"
```

```
main()
```

```
{
```

```
    int i,j,k;
```

```
    while (1)
```

```
    {
```

```
        VLED (Green, OFF);
```

```
        for (i=0;i<500000;i++)
```

```
        {
```

```
            VLED (Green, ON);
```

```
            for (i=0;i<500000;i++)
```

```
            {
```

```
                k=j;
```

```
            }
```

```
    }
```

```
VLED (color, state)
```

```
int    color, state;
```

```
{
```

```
volatile unsigned char *ptr;
```

```
const  int    addr = CSRC_ADDR;
```

```
    unsigned char    data;
```

```
    ptr = (unsigned char *) addr;
```

```
    data = *ptr;
```

```
    if (color == Green)
```

```
        data = (data & 0xbf) | (state << 6);
```

```
    else /* Yellow */
```

```
        data = (data & 0x7f) | (state << 7);
```

```
    *ptr = data;
```

```
    /* write with LED
```

```
    */
```

```
}
```





## pass1.ld

```
/* command file for "pass 1" of building a "linear" system.
*/
```

## MEMORY

```
{
    l_reg_0 : org = 0x00000000, len = 0x10000
    l_reg_1 : org = 0x40000000, len = 0x10000
    l_reg_2 : org = 0x80000000, len = 0x10000
    l_reg_3 : org = 0xc0000000, len = 0x10000
}
```

## SECTIONS

```
{
    .text :
    {
        } > l_reg_0
    GROUP :
    {
        .data : {}
        .bss : {}
    } > l_reg_1
    vreg2 :
    {
        initial_frame.o
    } > l_reg_2
    reg3 :
    {
        _intr_stack = .;
        . += 0x1000; /* reserve one page for intr. stack */
        f_handle.o
        f_table.o
        i_handle.o
        i_table.o
    } > l_reg_3
}
```

## pass1a.ld

```
/* command file for "pass 1" of building a "linear" system.
*/
```

## MEMORY

```
{
    l_reg_0 : org = 0x00000000, len = 0x10000
    l_reg_1 : org = 0x40000000, len = 0x10000
    l_reg_2 : org = 0x80000000, len = 0x10000
}
```

## SECTIONS

```
{
    .text :
    {
        } > l_reg_0
    GROUP :
    {
        .data : {}
        /*.=align(0x10); */
        .bss : {}
    } > l_reg_1
    vreg2 :
    {
        initial_frame.o
    } > l_reg_2
}
```

## pass2.ld

```

/* command file for "pass 2" of building a ROM system
 * without RTK.
 */

MEMORY
{
    image : origin = 0x0, length = 0x400000
}

SECTIONS
{
    GROUP :
    {
        startup :      {
            startup.o
            fix_pte.o
            . = align(0x1000);
        }
        proc1 :        {
            . = align(0x1000);
            _p1_reg_0_PA = .;
            lin1 (.text)
            . = align(0x1000);
            _p1_reg_1_PA = .;
            lin1 (.data)
            lin1 (.bss)
            . = align(0x1000);
            _p1_reg_2_PA = .;
            lin1 (vreg2)
            . = align(0x1000);
            _reg_3_PA = .;
            lin1 (reg3)
            . = align(0x1000);
            _reg_3_end = .;

            _p1_reg_0_len = _p1_reg_1_PA - _p1_reg_0_PA;
            _p1_reg_1_len = _p1_reg_2_PA - _p1_reg_1_PA;
            _p1_reg_2_len = _reg_3_PA - _p1_reg_2_PA;
            _reg_3_len = _reg_3_end - _reg_3_PA;
        }
        proc2 :        {
            . = align(0x1000);
            _p2_reg_0_PA = .;
            lin2 (.text)
            . = align(0x1000);
            _p2_reg_1_PA = .;
            lin2 (.data)
            lin2 (.bss)
            . = align(0x1000);
            _p2_reg_2_PA = .;
            lin2 (vreg2)
            . = align(0x1000);
            _p2_reg_2_end = .;

            _p2_reg_0_len = _p2_reg_1_PA - _p2_reg_0_PA;
            _p2_reg_1_len = _p2_reg_2_PA - _p2_reg_1_PA;
            _p2_reg_2_len = _p2_reg_2_end - _p2_reg_2_PA;
        }
    }
}

csl = -(seg_table_ptr + prcb_ptr + start_ip);

```



---

*Appendix E*  
*Considerations for Writing*  
*Portable Software*

---



## APPENDIX E CONSIDERATIONS FOR WRITING PORTABLE SOFTWARE

This appendix describes those parts of the 80960MC design that are implementation dependent. This information is provided to facilitate the design of programs and kernel code that will be portable to other implementations of the 80960MC architecture.

### ARCHITECTURE RESTRICTIONS

The following aspects of the 80960MC processor's operation are deviations from the 80960MC architecture:

1. Only the low-order 16 bits of the next-time-slice and residual-time-slice fields in the PCB are used. The upper 16 bits are ignored.
2. The minimum value that can be placed in the next-time-slice field is 16 (ticks). Assigning it a value less than 16 can result in endless loops.
3. When the addressing mode is set to physical, the **inspace** and **ldphy** instructions have an undefined effect.
4. On all bus write operations except those of the **synmov**, **synmovl**, and **synmovq** instructions, the processor ignores the BADAC pin (i.e., errors signaled on "normal" writes are ignored).
5. The check for out-of-range input values for the **expr**, **expri**, **logepr**, and **logepri** instructions is omitted; out-of-range inputs yield an undefined result.
6. Bits 5 and 6 of a machine-level instruction word in the REG and MEMB formats and bits 0 and 1 of the CTRL format are provided to designate special function registers. The 80960MC processor has no special function registers.
7. The 80960MC processor does not guarantee that the value in register r2 of the current frame is predictable.
8. (The following is a note rather than a restriction.) When using the REG-format instructions, the m bit for every operand that is not defined by the instruction should be set (e.g., code the unused operand as an arbitrary literal). This practice may reduce overhead in some situations.

### SALIGN PARAMETER

Stack frames in the 80960MC architecture are aligned on (SALIGN\*16) byte boundaries. SALIGN is an implementation defined parameter. For the 80960MC processor, SALIGN is 4. Stack frames for this processor are thus aligned on 64 byte boundaries.

The low-order N bits of the FP are ignored and always interpreted to be zero. The N parameter is defined by the following expression:  $SALIGN*16 = 2^N$ . Thus for the 80960MC processor, N is 6.



## BOUNDARY ALIGNMENT

The physical-address boundaries on which an operand begins has an impact on processor performance. For the 80960MC processor, the following is true:

- An operand that spans more word boundaries than necessary (e.g., addressing a 32-bit operand on a nonword boundary) suffers a moderate cost in speed because of extra bus and memory cycles.
- An operand that spans a 16-byte boundary suffers a large cost in speed.
- String operands that begin on nonword boundaries suffer a moderate cost in speed. String operands that begin on word boundaries but not on 16-byte boundaries suffer a small cost in speed.

## FAULTS

As described in Chapter 12, the processor enters the stopped state when a fault is detected while trying to invoke a procedure as the result of a system-error interrupt. When the processor enters the stopped state in this circumstance, it asserts the FAILURE pin.

The size of resumption records conditionally placed on the stack during faults and interrupts is 16 bytes.

## PHYSICAL MEMORY

The upper 16M bytes of physical memory are reserved for special functions of local-bus components, IACs, and the BXU.

## IACS

The mechanism for sending, receiving, and handling IAC messages is not defined in the 80960MC architecture. It is a special implementation of the 80960MC processor.

The write-external-priority flag in the processor controls is not defined in the 80960MC architecture.

## TIMING

A tick is defined for the 80960MC processor as 256 external clock periods (128 internal clock periods). Thus, for a 16-MHz processor (32-MHz external clock), a tick is 8 microseconds. For a 20-MHz processor, a tick is 6.4 microseconds.

The frequency at which an idle processor checks the dispatch port is implementation dependent. For the 80960MC processor, it is approximately once every tick.

The frequency at which a processor updates the idle-time field in the processor controls when it is counting idle time is also implementation dependent. For the 809BASE processor, it is approximately once every 32 ticks.

When the processor is spinning on a lock (e.g., when executing a **send**, **receive**, or **signal** instruction), the frequency at which the processor tries the lock is implementation dependent. For the 80960MC processor, it is once every tick until it is able to lock it. Provided that the execution timer and end-of-time-slice event are enabled, the process may eventually be suspended. When redispached, it will resume execution within the instruction and the locking operation will be retried. In the other circumstances where a processor needs to lock a data structure and it is already locked, it will try the lock approximately once every tick until it can lock the data structure.

## INTERRUPTS

The interrupt IAC message, the interrupt pins, and the interrupt register are not defined in the 80960MC architecture. They are special implementations for the 80960MC processor.

## INITIALIZATION

The 80960MC architecture does not define an initialization mechanism. The initialization mechanism and procedures described in this manual are implementation dependent for the 80960MC processor.

## MULTIPROCESSOR PREEMPTION

The multiprocessor preemption mechanism described in Chapter 15 is implementation dependent for the 80960MC processor. Also, the write external priority flag and the interim priority field in the processor controls are implementation dependent.

## BREAKPOINTS

The breakpoint registers in the 80960MC processor are not defined in the 80960MC architecture.

## IMPLEMENTATION DEPENDENT INSTRUCTIONS

The **synmov**, **synmovl**, **synmovq**, and **synld** instructions are not defined in the 80960MC architecture and are implementation dependent in the 80960MC processor.

## LOCK PIN

The  $\overline{\text{LOCK}}$  pin is not defined in the 80960MC architecture and is implementation dependent in the 80960MC processor.

When the processor is spinning on a lock (e.g., when executing a send, receive, or signal instruction), the frequency at which the processor tries the lock is implementation dependent. For the 80960MC processor, it is once every tick until it is able to lock it. Provided that the execution timer and end-of-time-slice event are enabled, the process may eventually be suspended. When redispached, it will resume execution within the instruction and the locking operation will be retried. In the other circumstances where a processor needs to lock a data structure and it is already locked, it will try the lock approximately once every tick until it can lock the data structure.

## INTERRUPTS

The interrupt IAC message, the interrupt pins, and the interrupt register are not defined in the 80960MC architecture. They are special implementations for the 80960MC processor.

## INITIALIZATION

The 80960MC architecture does not define an initialization mechanism. The initialization mechanism and procedures described in this manual are implementation dependent for the 80960MC processor.

## MULTIPROCESSOR PREEMPTION

The multiprocessor preemption mechanism described in Chapter 12 is implementation dependent for the 80960MC processor. Also, the write external priority flag and the interim priority field in the processor controls are implementation dependent.

## BREAKPOINTS

The breakpoint registers in the 80960MC processor are not defined in the 80960MC architecture.

## IMPLEMENTATION DEPENDENT INSTRUCTIONS

The `syncmov`, `syncmovb`, and `syncid` instructions are not defined in the 80960MC architecture and are implementation dependent in the 80960MC processor.

## LOCK PIN

The `LOCK` pin is not defined in the 80960MC architecture and is implementation dependent in the 80960MC processor.

---

# *Index*

---

---

Index

---

# INDEX

M82965 15-3, 15-5  
80960 Architecture  
implementation dependent aspects of  
80960MC processor E-1  
M82965 2-7  
80960 Architecture  
debugging and monitoring 2-5  
efficient interrupt model 2-3  
efficient procedure call mechanism 2-4  
extensions included in 80960MC proces-  
sor 2-5  
fault handling capability 2-4  
instruction cache 2-2  
load and store model 2-2  
local register sets 2-2  
overview of 2-1  
parallel instruction execution 2-3  
register scoreboarding 2-3  
simplified programming environment  
2-4  
single clock instructions 2-3  
special function registers 2-5  
versatile instruction set and addressing  
2-4

## A

Abase 5-7  
Absolute addressing mode, description of  
5-7  
AC.cc 17-3  
Access status field 8-10  
Accessed flag 8-10  
Add instructions 6-8  
Add with Carry Instruction 6-8  
**addc** 6-8, 17-6  
**addi, addo** 6-8, 17-7  
**addr, addr1** 7-17, 17-8  
addr, notation 17-2

Address space  
address space boundaries 8-25  
of process 13-1  
partitioning of 3-10  
process state 13-1  
regions 3-10  
typical use of 3-10  
Address translation modes  
addressing mode flag 9-7  
changing 9-12  
consequences of changing 9-12  
description of 9-12  
physical addressing mode 9-14  
physical vs. virtual 8-1  
treatment of SS's 9-12  
virtual addressing mode 9-14  
Addressing mode flag 8-1, 9-7  
Addressing modes, used in instructions  
abase 5-7  
absolute 5-7  
description of 5-6  
index 5-7  
index with displacement 5-7  
IP with displacement 5-7  
register indirect 5-7  
register indirect with index 5-7  
scale factor 5-7  
**alterbit** 6-10, 7-15, 17-10  
Altered flag 8-10  
**and, andnot** 6-10, 17-11  
Architecture  
See 80960 Architecture  
Arithmetic controls  
arithmetic status field 3-8  
changing of 13-9  
condition code flags 3-7  
description of 3-6  
fault masks and flags 12-12  
floating-point flags and masks 3-9  
floating-point normalizing mode flag  
3-9

floating-point rounding control field 3-9

functions of bits 3-7

in PCB 3-6, 13-5

initializing 3-6

integer-overflow flag and mask 3-8

modify arithmetic controls instruction 6-17

modifying 3-6

no imprecise faults flag 3-9, 12-22

process state 13-1

saving and restoring 3-7

structure of 3-6

use with conditional receive 14-17

Arithmetic faults 12-25

Arithmetic status field 7-11, 7-17

description of 3-8

Arithmetic zero-divide fault 12-2, 12-25, 17-59, 17-64, 17-90

**atadd** 6-6, 6-16, 15-7, 17-12

**atanr, atanrl** 7-18, 17-13

**atmod** 6-6, 6-16, 15-7, 17-15

Atomic operations

atomic instructions 6-16, 15-6

description of 8-2

Automatic process dispatching

description of 14-10

dispatching action 14-10

process suspension 14-11

scheduling instructions 14-10

self dispatching 2-7

time slice scheduling 14-11

## B

**b** 6-13, 17-16

Bad access fault 12-2, 12-31

**bal, balx** 4-15, 6-13, 16-4, 17-18

**bbc, bbs** 6-14, 17-20

BCL C-2

**be, bg, bge** 6-13, 17-22

Biased exponent 7-3, 7-4

Bipaged region segment descriptor 8-12

## Bits and bit fields

bit addressing 5-5

bit field instructions 6-11

bit operation instructions 6-10

description of 5-4

**bl, ble, bne** 6-13, 17-22

**bno, bo** 6-13, 7-17, 17-22

## Branch and link

description of 4-15

instructions 6-13

## Branch trace

event flag 16-2

fault 12-2, 12-37

mode 16-4

mode flag 16-2

## Breakpoint registers

description of 16-5, 16-6

set breakpoint register IAC 11-20, 16-5

## Breakpoint trace

event flag 16-2

fault 12-2, 12-37, 17-73, 17-88

mode 16-5

mode flag 16-2

## Bus control logic

See BCL

## Bus extension unit

See M82965

**bx** 6-13, 17-16

Byte addressing 5-5

Byte string, description of 5-4

## C

### Caching of memory accesses

cacheable flag 8-10

description of 8-3

**call** 4-8, 6-15, 12-11, 16-4, 17-25

Call instructions 6-15

### Call trace

event flag 16-2

fault 12-2, 12-37

mode 16-4



mode flag 16-2

**calls** 4-10, 4-14, 6-15, 12-7, 14-18, 16-3,  
16-5, 17-27

**callx** 4-8, 6-15, 12-7, 12-11, 16-4, 17-29

Check bit and branch instructions 6-14

Check dispatch port flag 9-7, 10-10, 14-15

Check process notice IAC 11-5, 12-13,  
12-28, 13-6, 13-9

Check-sum words 9-17, 9-21

**chkbit** 6-10, 7-15, 17-31

**classr, classrl** 7-11, 7-17, 7-20, 17-32

Clear, definition of 1-4

**clrbt** 6-10, 17-34

**cmpdeci, cmpdeco** 6-12, 17-36

**cmpi** 6-11, 17-35

**cmpibe, cmpibne, cmpibl, cmpible,**  
**cmpibg, cmpibge, cmpibo,**  
**cmpibno** 6-14, 17-44

**cmpinci, cmpinco** 6-12, 17-37

**cmpo** 6-11, 17-35

**cmpobe, cmpobne, cmpobl, cmpoble,**  
**cmpobg, cmpobge** 6-14, 17-44

**cmpor, cmporl** 7-17, 17-38

**cmpr, cmprl** 7-17, 17-40

**cmpstr** 6-19, 17-42

Communication port  
description of 14-1, 14-10, 14-15  
instructions 6-18  
send service instruction 14-10  
structure of 14-16  
use of 14-12

Compare and branch instructions 6-14

Compare and decrement instructions 6-12

Compare and increment instructions 6-12

Compare instructions 6-11

**concmpi, concmpo** 6-11, 17-47

Condition code  
See Condition code flags

Condition code flags  
description of 3-7  
in floating-point compare instructions  
7-17

in floating-point operations 7-11, 7-17

in test instructions 6-14

modification of 6-17

condition code scoreboarding C-12

Conditional branch instructions 6-13

Conditional compare instructions 6-11

**condrec** 6-18, 10-5, 14-17, 17-48

**condwait** 6-18, 14-13, 17-50

Constraint faults 12-26

Constraint range fault 12-2, 12-26, 17-69

Contents fault 12-2, 12-39

Continue initialization IAC 11-6

Control fault 12-2, 12-36

**cosr, cosrl** 7-18, 17-52

**cpysre, cpysre** 7-15, 7-20, 17-54

Current process SS 9-8

**cvtilr, cvtir** 7-16, 17-55

**cvtri, cvtril, cvtzri, cvtzril** 7-16, 17-56

## D

**daddc** 6-19, 17-58

Data length conversion 6-11

Data structures, quick reference A-12

Data types  
bits and bit fields 5-4  
byte string 5-4  
decimal 5-3  
description of 5-1  
integer 5-1  
ordinal 5-1  
quad word 5-5  
real 5-2  
triple word 5-5

Debugging support  
overview of 2-5  
See also Tracing

Decimal Multiplication and Division 6-20

Decimals  
data type 5-3  
instructions 6-19  
multiplication and division 6-19

Denormalized numbers  
  definition of 7-5  
  denormalization technique 7-5

Descriptor faults 12-27

disp, notation 17-2

Dispatch fault 12-2, 12-36

Dispatch port  
  assigned to processor 14-10  
  assignment of process to 14-10  
  check dispatch port flag in processor controls 9-7  
  description of 14-1, 14-9  
  dispatch port SS in PRCB 9-8  
  high-level process management facilities 14-1  
  in multiprocessor systems 15-4  
  pointer for in PCB 13-6  
  structure of 14-10

**divi, divo** 6-8, 17-59

Divide instructions 6-8

**divr, divrl** 7-17, 17-60

**dmovt** 6-19, 17-62

**dsubc** 6-19, 17-63

## E

**ediv** 6-8, 17-64

efa, notation 17-2

**emul** 6-8, 17-65

Event fault  
  event fault request flags 11-5, 12-12, 12-13, 13-6  
  event notice fault 11-5, 12-2, 12-13, 12-28  
  reference information 12-28

Exceptions, floating-point  
  See Floating point faults

Execution environment  
  address space 3-1, 3-2  
  arithmetic controls 3-6  
  description of 3-1  
  floating-point registers 3-4  
  global registers 3-4

instruction cache 3-12

instruction pointer 3-5

local registers 3-4

process controls 3-10

trace controls 3-10

Execution mode  
  description of 4-13  
  execution mode flag 4-5, 13-4

Execution time field 13-7

Exponent, in floating point format 7-2

**expr, exprl** 7-19, 17-66

Extended multiply and divide instructions 6-8

External IACs  
  See IACs

**extract** 6-11, 17-68

**F**

FAILURE pin 9-21

Fault handling  
  aborting a process 12-11  
  control flags and masks 12-12  
  fault handler, description of 12-1  
  fault handler, procedures 12-8  
  fault handling actions 12-16  
  fault handling methods 12-3  
  local calls to fault handling procedures 12-7  
  overview of fault-handling facilities 12-1  
  possible fault-handler actions 12-9  
  procedure table calls to fault handling procedures 4-11, 12-7  
  process and instruction resumption following a fault 12-9  
  returning from a fault with resumption 12-10  
  returning from a fault without resumption 12-11  
  software requirements for handling faults 12-5  
  support for 2-4  
  trace fault handling 12-7, 12-8

See also Fault record, Fault table, Faults

#### Fault record

- description of 12-14
- location of fault record 12-15
- location of resumption record 12-15
- resumption record 12-15
- saved instruction pointer 12-15

#### Fault table 12-3

- description of 9-4, 12-5
- fault table entries 12-7
- fault table pointer in PRCB 9-9, 12-7
- location of in memory 12-7
- required at initialization 9-14, 9-21

#### Fault table pointer 9-9

#### Fault tolerance, support for 2-7

#### Fault-if instructions 12-13

**faulte, faultne, faultl, faultle, faultg,  
faultge, faulto, faultno** 6-15,  
17-69

#### Faults

- arithmetic faults 12-25
- constraint faults 12-26
- description of 9-5
- descriptor faults 12-27
- event faults 12-28
- event notice fault 12-13
- fault instructions 6-15, 12-13
- floating-point faults 12-29
- generating a fault 12-13
- halt 12-5
- halt action 12-22
- interrupts and faults 12-13
- location of resumption record 12-15
- machine faults 12-31
- multiple fault conditions 12-5
- operation faults 12-32
- override fault-handling action 12-20
- overrides 12-4
- precise and imprecise faults 12-22
- process faults 12-33
- process state after a fault 12-9
- protection faults 12-34
- refault flag 13-5

reference information on faults 12-24

resumption record 12-15

resumption record in PCB 13-6

saved instruction pointer 12-15

saved process controls 12-19

standard faults 17-3

structural faults 12-36

system error interrupt 12-4, 12-21

trace faults 12-37

type faults 12-39

types and subtypes 12-1

virtual memory faults 12-40

See also Fault handling, Fault record

#### FIFO port 14-7

conditional receive message mechanism  
14-17

description of 14-7

lock field 14-7

queue head SS 14-7

queue state flag 14-7

queue tail SS 14-7

receive message mechanism 14-17

send message mechanism 14-16

send service mechanism 14-18

#### fill 6-19, 17-71

flit, notation 17-2

Floating inexact fault 7-25, 7-26, 12-2,  
12-29, 17-8, 17-13, 17-52, 17-55,  
17-66, 17-80, 17-82, 17-85, 17-97,  
17-101, 17-112, 17-119, 17-121,  
17-134, 17-137, 17-143, 17-151

Floating inexact flag and mask 7-11, 7-25,  
12-12

Floating invalid-operation fault 7-23, 12-2,  
12-29, 17-8, 17-13, 17-38, 17-40,  
17-52, 17-60, 17-66, 17-80, 17-82,  
17-85, 17-97, 17-101, 17-112,  
17-119, 17-121, 17-134, 17-137,  
17-143, 17-151

Floating invalid-operation flag and mask  
7-11, 7-20, 7-23, 12-12

Floating overflow fault 7-24, 12-2, 12-29,  
17-8, 17-60, 17-82, 17-85, 17-97,  
17-101, 17-112, 17-119, 17-121,

- 17-137, 17-143, 17-151
- Floating overflow flag and mask 7-11,  
7-24, 7-25, 12-12
- Floating point
  - architecture support for 7-1
  - arithmetic controls 7-11
  - arithmetic vs. non-arithmetic instructions 7-20
  - basic arithmetic instructions 7-17
  - biased exponent 7-3, 7-4
  - branch instructions 7-17
  - classification instructions 7-17
  - comparison instructions 7-17
  - data movement instructions 7-15
  - data type conversion 7-15
  - denormalized numbers 7-5
  - execution environment for floating-point operations 7-7
  - exponent 7-2
  - exponential instructions 7-19
  - finite values 7-4
  - floating inexact exception 7-25
  - floating invalid operation exception 7-23
  - floating overflow exception 7-24
  - floating reserved encoding exception 7-22
  - floating underflow exception 7-24
  - floating zero-divide exception and fault 7-23
  - format of binary floating-point numbers 7-2
  - fraction 7-2
  - IEEE standard 7-1, 7-2, 7-4, 7-6, 7-7,  
7-14, 7-17, 7-19
  - infinities 7-6
  - instruction format 7-14
  - instruction operands 7-14
  - integer 7-2
  - j-bit 7-2
  - literals 7-14
  - loading and storing floating-point values 7-9
  - logarithmic instructions 7-19
  - moving floating-point values 7-10
  - NaNs 7-4, 7-20
  - normalized number 7-3
  - normalizing mode 7-12
  - pi 7-18
  - real data types 5-2, 7-7
  - real number and NaN encodings 7-4,  
7-7
  - real number formats 7-7
  - real number notation 7-3
  - real number system 7-1
  - register alignment for floating-point values 7-9
  - registers, storage of floating-point numbers in 7-8
  - rounding control 7-12
  - scale instructions 7-19
  - sign bit 7-2
  - significand 7-2
  - summary of floating-point instructions 7-15
  - support for 2-5
  - trigonometric instructions 7-18
  - underflow condition 7-26
  - zeros 7-4
  - See also Floating point faults
- Floating point faults 12-29
  - exceptions 7-6, 7-21
  - fault handling 7-21, 7-22
  - floating inexact exception 7-21
  - floating invalid operation exception 7-21
  - floating overflow exception 7-21
  - floating reserved encoding exception 7-21
  - floating underflow exception 7-21
  - floating zero divide exception 7-21
  - override flags 7-24, 7-25
- Floating point unit
  - See FPU
- Floating reserved-encoding fault 7-22,  
12-2, 12-29, 17-8, 17-13, 17-38,  
17-40, 17-52, 17-54, 17-60, 17-66,  
17-80, 17-82, 17-85, 17-97,  
17-101, 17-112, 17-119, 17-121,

- 17-134, 17-137, 17-143, 17-151
  - Floating underflow fault 7-25, 7-26, 12-2, 12-29, 17-8, 17-13, 17-60, 17-66, 17-80, 17-82, 17-85, 17-97, 17-101, 17-112, 17-119, 17-121, 17-134, 17-137, 17-143, 17-151
  - Floating underflow flag and mask 7-11, 7-24, 12-12
  - Floating zero-divide fault 7-23, 12-2, 12-29, 17-60, 17-80, 17-85, 17-112, 17-121
  - Floating zero-divide flag and mask 7-11, 7-23, 12-12
  - Floating-point flags and masks 3-9
  - Floating-point normalizing mode flag 3-9, 7-11, 7-12
  - Floating-point registers
    - description of 3-4
    - field in PCB 13-6
    - register model 3-2
    - storage of 3-4
    - See Registers
  - Floating-point rounding control field 3-9, 7-11
  - Flush local registers
    - flush local registers IAC 4-7, 11-7
    - instruction 6-16
  - Flush process IAC 11-8
  - Flush TLB IAC 11-9
  - Flush TLB page table entry IAC 11-10
  - Flush TLB physical page IAC 11-11
  - Flush TLB segment entry IAC 11-12
  - flushreg** 4-7, 6-16, 17-72
  - fmark** 6-16, 12-13, 16-1, 16-5, 16-6, 16-7, 17-73
  - Force mark instruction 6-16, 12-13
  - FP, frame pointer 3-4, 4-14
    - description of 4-3
    - location at initialization 9-15
  - FPU C-8
  - Fraction, in floating-point format 7-2
  - Frame pointer
    - See FP
  - Frame return status field 10-9
  - Freeze IAC 11-13
  - freg, notation 17-2
- ## G
- Global registers
    - description of 3-4
    - field in PCB 13-6
    - FP 3-4
    - process state 13-1
    - register alignment 3-4
    - register model 3-2
    - storage of 3-4
    - storing of RIP on a branch and link instruction 4-15
- ## I
- IAC fault 12-2, 12-36
  - IAC pin 10-7, 15-2
  - IACs
    - check process notice IAC 11-5, 12-13, 12-28
    - continue initialization IAC 11-6
    - description of 9-4
    - external IAC message format 15-1
    - external IACs 11-1, 15-1
    - faults 11-3
    - flush local registers IAC 11-7
    - flush process IAC 11-8
    - flush TLB IAC 11-9
    - flush TLB page table entry IAC 11-10
    - flush TLB physical page IAC 11-11
    - flush TLB segment entry IAC 11-12
    - freeze IAC 11-13
    - IAC fault 12-2, 12-36
    - IAC pin 15-2
    - internal IACs 11-1
    - interrupt IAC 11-14
    - introduction to 11-1
    - mechanisms for exchanging 11-1
    - message, description of 11-1

- message, format of 11-2
- modify processor controls IAC 11-15
- preempt process IAC 11-16
- priorities 9-10
- purge instruction cache IAC 11-17
- receiving and handling external IACs 15-2
- receiving and handling internal IACs 11-3
- reference information 11-4
- reinitialize processor IAC 11-18
- restart processor IAC 11-19, 12-22
- sending external IACs 15-1
- sending internal IACs 11-3
- set breakpoint register IAC 11-20
- software requirements for handling internal IACs 11-1
- stop processor IAC 11-21
- store processor IAC 11-22
- store system base IAC 11-23
- summary of IACs 11-2
- test pending interrupts IAC 11-24
- warmstart processor IAC 11-25
- ID C-4
- Idle time
  - idle time field 9-9, 9-13
  - idle timing 9-13
- IEU C-6
- IFU C-3
- Index with displacement addressing mode, description of 5-7
- Index, description of 5-7
- Indivisible, description of 8-2
- Inexact result, definition of 7-12
- Initial memory image, description of 9-17
- Initialization code example D-1
- Initialization of the processor
  - Building a memory image 9-19
  - check-sum words 9-17
  - continue initialization IAC 11-6
  - description of 9-15
  - first stage of initialization 9-21
  - initial memory image 9-14, 9-17
  - initialization code 9-19
  - initialization code example D-1
  - initialization fault table 9-21
  - initialization heap 9-21
  - initialization interrupt table 9-21
  - initialization page tables 9-20
  - initialization PCB 9-20
  - initialization PRCB 9-19, 9-20
  - initialization segment table 9-17, 9-19
  - initialization stack 9-21
  - internal PCB fields 13-8
  - reading the PRCB 9-9
  - reinitialize processor IAC 11-18
  - required PRCB for single-task system 9-15
  - restart processor IAC 11-19
  - second stage of initialization 9-23
  - segment table pointer 9-3
  - self test 9-21
  - typical initialization scenario 9-21
  - warmstart processor IAC 11-25
- Initialization segment table
  - description of 9-17
- inspacc 6-20, 17-74
- Inspect access instruction 6-20
- Instruction cache
  - description of 2-2, 3-12, C-3
  - purge instruction cache IAC 11-17
- Instruction decoder
  - See ID
- Instruction execution unit
  - See IEU
- Instruction fetch unit
  - See IFU
- Instruction list 9-2
- Instruction pointer
  - See IP
- Instruction reference
  - introduction to 17-1
  - Notation 17-1
- Instruction suspension
  - description of 9-13
  - resumption record field in PRCB 9-9



- Instruction timing
  - bit instructions C-10
  - branch instructions C-12
  - call and return instructions C-13
  - decimal instructions C-17
  - description of C-8
  - floating point instructions C-17
  - integer and ordinal arithmetic instructions C-11
  - load instructions C-14
  - logical instructions C-9
  - miscellaneous complex instructions C-14
  - multiply and divide instructions C-12
  - register move instructions C-10
  - store instructions C-16
- Instruction trace
  - event flag 16-2
  - fault 12-2, 12-37
  - mode 16-4
  - mode flag 16-2
- Instructions
  - arithmetic 6-6
  - assembly-language format 6-1
  - bit and bit field 6-10
  - branch 6-12
  - call and return 6-15
  - comparison 6-11
  - data length conversion 6-11
  - data movement 6-4
  - debug 6-16
  - decimal 6-19
  - detailed reference information 17-1
  - extended arithmetic 6-8
  - fault instructions 6-15
  - instruction groups 6-2
  - logical 6-10
  - machine-level instruction formats B-1
  - process management 6-17
  - processor management 6-16
  - quick reference A-1
  - string 6-19
  - summary of 80960MC instruction-set extensions 6-3
  - summary of 80960 instructions 6-2
  - See also Machine-level formats
- INT0, INT1, INT2, INT3 pins 10-6, 10-7
- INTA pin 10-7
- Integer overflow
  - description of 3-8
  - fault 12-2, 12-9, 12-25, 17-7, 17-56, 17-59, 17-100, 17-111, 17-131, 17-139, 17-142
  - flag 3-8, 7-11, 12-12, 12-25
  - mask 3-8, 7-11, 12-12, 12-25
- Integer, description of 5-1
- Interagent communication messages
  - See IACs
- Interim priority field 9-8, 15-5
- Internal state field, of process controls 12-11
- Interprocess communication
  - instructions 6-18
  - support for 2-7
  - See Messages passing
- Interrupt control register
  - addresses mapped to in physical memory 10-7
  - description of 10-6
  - uses of 10-6
- Interrupt handler
  - used for initialization 9-22
- Interrupt handling
  - interrupt control register 10-6
  - interrupt handler procedures 10-4
  - interrupt stack 10-5
  - interrupt table 10-2
  - interrupt table sharing 10-4
  - location of interrupt handler procedures 10-4
  - restrictions on interrupt handler 10-5
  - software requirements for interrupt handling 10-1
  - support for 2-3
- Interrupt IAC 10-13, 11-14
  - description of 10-7



- Interrupt pins
  - description of 10-6
  - uses of 10-6
- Interrupt record
  - description of 10-10
- Interrupt stack
  - description of 9-3, 10-5
  - interrupt stack pointer in PRCB 9-8
  - required at initialization 9-14
- Interrupt stack pointer 9-8
- Interrupt table
  - description of 9-3, 10-2
  - interrupt table pointer in PRCB 9-8
  - interrupt table sharing 10-4
  - required at initialization 9-14
- Interrupt table pointer 9-8
- Interrupt vectors, description of 10-2
- Interrupts
  - description of 9-4
  - idle state interrupt 10-10
  - idle-interrupted state interrupt 10-12
  - in a multiprocessor system 15-7
  - interrupt control register 15-2
  - interrupt handling actions 10-8
  - interrupt IAC 10-7, 11-14
  - interrupt pins 10-6
  - interrupt record 10-10
  - overview of interrupt facilities 10-1
  - pending interrupts 10-12
  - priorities 9-10, 10-2
  - process executing state interrupt 10-9
  - process interrupt state interrupt 10-10
  - servicing an interrupt 10-8
  - signaling interrupts 10-6
  - system-error interrupt 9-9, 10-8, 12-3, 12-4, 12-5, 12-16, 12-21, 12-22
  - system-error interrupt vector 12-4
  - test pending interrupts IAC 11-24, 15-7
  - vectors 10-2
  - See also Interrupt handling
- INTR pin 10-7
- Invalid descriptor fault 8-22, 12-2, 12-27, 17-74
- Invalid opcode fault 12-2, 12-32
- Invalid operand fault 12-2, 12-32
- Invalid PTDE fault 8-10, 8-24, 12-2, 12-40
- Invalid PTE fault 8-10, 8-22, 8-24, 12-2, 12-40
- Invalid segment descriptor 8-16
- Invalid segment descriptor fault 8-21, 8-22
- Invalid segment table entry fault 8-10, 12-2, 12-40
- Invalid SS fault 12-2, 12-26
- IP
  - description of 3-5
  - procedure table entry 4-11
  - storage of 3-5
- IP with displacement addressing mode 5-7
- J**
  - J-bit 7-2
- K**
  - Kernel 1-1
    - altering process controls 13-8
    - process scheduling in multiprocessor system 15-4
    - supervisor procedure 4-13, 4-14
- L**
  - Large segment table segment descriptor 8-14
  - ld, ldib, ldis, ldl, ldob, ldos, ldq, ldt** 5-5, 6-5, 7-9, 17-75
  - lda** 3-5, 6-6, 17-77
  - ldphy** 6-20, 8-25, 17-78
  - ldtime** 6-17, 10-5, 14-6, 17-79
  - lit, notation 17-2
  - Literal
    - description of 5-5
    - floating-point 7-14
    - ordinal 5-5
  - Load address instruction 6-6
  - Load instructions 6-5

Load physical address instruction 6-20

Local call

- call operation 4-8
- description of 4-8
- return operation 4-8

Local registers

- call/return mechanism 4-1
- description of 2-2, 3-4
- PFP 3-4
- process state 13-1
- purpose of 3-4
- register alignment 3-4
- register model 3-2
- relationship to procedure stack 4-3
- RIP 3-4
- SP 3-4
- stack-frame cache 4-3

LOCK line 8-2

**logbnr, logbnrl** 7-19, 17-80

**logepr, logeprl** 7-19, 17-82

Logical instructions 6-10

**logr, logrl** 7-19, 17-85

**M**

Machine faults 12-31

Machine-level formats 6-1, B-1

Manual

- guide to 1-1
- structure of 1-1

**mark** 6-16, 12-13, 16-1, 16-5, 16-6, 16-7, 17-88

Mark instruction 6-16, 12-13

mem, notation 17-2

Memory management facilities, introduction to 8-1

Memory management unit

- See MMU

Messages and message passing

- applications of messages 14-19
- communication port 14-15
- current port or semaphore SS 14-9
- high-level process management facilities 14-1

interprocessor communication 14-15

- kernel support for message passing 14-18
- link SS 14-9
- mechanism for interprocess communication 14-16
- message field in PCB 13-6
- message, description of 14-7, 14-9
- receive message mechanism 14-17
- send message mechanism 14-16
- send service mechanism 14-18

Micro-instruction sequencer

- See MIS

MIS C-6

MMU C-1

Mnemonic 17-2

**modac** 3-6, 6-17, 17-89

**modi** 6-8, 17-90

**modify** 6-11, 17-91

Modify process controls instruction 6-16, 6-18

Modify processor controls IAC 9-10, 9-12, 11-15

Modify trace controls instruction 6-16

**modpc** 6-16, 10-13, 13-8, 14-5, 15-5, 17-92

**modtc** 6-16, 16-2, 17-94

Modulo instructions 6-8

**mov, movl, movq, movt** 5-5, 6-6, 7-10, 7-15, 17-95

Move instructions 6-6

**movqstr** 6-19, 17-96

**movr, movre, movrl** 7-9, 7-10, 7-15, 7-20, 17-97

**movstr** 6-19, 17-99

**muli, mulo** 6-8, 17-100

**mulr, mulrl** 7-14, 7-17, 17-101

Multiple processor operation

- See Multiprocessing

Multiply instructions 6-8

**Multiprocessing**  
atomic instructions 15-6  
description of 9-1  
dispatch port 15-4  
external IACs 15-1  
high-level process management facilities  
15-3  
interrupt handling 15-7  
memory management facilities 8-1  
overview of multiple processor support  
facilities 15-1  
preemption 15-4  
preemption action 15-6  
preemption control 15-4  
process scheduling and dispatching  
15-4  
receiving and handling external IACs  
15-2  
sending external IACs 15-1  
support for 2-7  
use of processes 13-2  
**Multiprocessor preempt flag** 9-5, 10-10,  
15-5  
**Multiprocessor preemption field in PRCB**  
9-9  
**Multitasking**  
description of 9-1  
memory management facilities 8-1  
priorities 9-10  
processes vs. tasks 9-1  
support for 2-6  
use of processes 13-2

## N

**nand** 6-10, 17-103  
**NaNs**  
arithmetic vs. non-arithmetic instructions  
7-20  
classify instructions 7-17  
comparison 7-17  
defined 7-6  
encodings 7-4, 7-7  
extended-real format 7-7

invalid-operation exception 7-23  
operations on 7-20  
QNaN 7-6, 7-17, 7-23  
QNaN, definition of 7-20  
rounding 7-13  
SNaN 7-6, 7-17, 7-23  
SNaN, definition of 7-20  
unordered 7-17  
unordered classification 3-8  
**Next time slice field** 13-7, 14-5  
**No imprecise faults flag** 3-9, 12-12, 12-22  
**Nonpreempt limit field** 9-7, 15-5  
**nor** 6-10, 17-104  
**Normalized number** 7-31  
**Normalizing mode, floating-point normaliz-  
ing mode flag** 3-9  
**not, notand** 6-10, 17-105  
**Notation** 1-3  
**notbit** 6-10, 17-106  
**notor** 6-10, 17-107

## O

**Operating-system kernel**  
See **Kernel**  
**Operation faults** 12-32  
**or, ornot** 6-10, 17-108  
**Ordinal, description of** 5-1  
**Override faults**  
See **Faults**

## P

**Padding area, description of** 4-5  
**Page rights**  
description of 8-20  
fault 8-24, 12-2, 12-34  
**Page Table and Page Table Directory**  
invalid page table (directory) entry  
8-20  
page rights 8-20  
page table directory entry 8-20  
page table entry 8-19  
structure of 8-18

- Paged region segment descriptor 8-12
- Paging
  - bipaged segment 8-16
  - overview of 8-16
  - page length 8-3
  - page table and page table directory structures 8-18
  - paged segment 8-16
  - paging method field 8-11
  - protection of pages 2-6
  - spanning page boundaries 8-25
  - unpaged segment 8-16
- Parameter passing
  - description of 4-9
  - in an argument list 4-9
  - through global registers 4-9
  - through the procedure stack 4-9
- PCB
  - arithmetic controls field 13-5
  - binding process to processor 14-2
  - current process SS 9-8
  - description of 9-3, 13-1, 13-2
  - dispatch port SS field 13-6, 14-10
  - event-fault request flags 12-13
  - execution time field 13-7
  - global registers field 13-6
  - lock field 13-6
  - low-level process management facilities 14-1
  - next time slice field 13-7
  - preempt flag 15-4
  - process controls 13-4
  - process notice field 11-5, 12-13, 13-6
  - process resumption following a fault 12-10
  - process state 13-1
  - queue record 13-6
  - received message field 13-6
  - region 0, 1, and 2 SS fields 13-6
  - relationship to process 9-1
  - required at initialization 9-14, 9-20
  - residual time slice field 13-7
  - resumption record field 9-9, 13-6
  - segment descriptor 8-13
  - storing of PCB fields in processor 13-7
  - trace controls field 13-5
  - See also Process, Process management
- Pending interrupts
  - checking for 10-13
  - handling of 10-13
  - posting of 10-12
  - servicing of 10-12
- PFP 3-4, 10-9
  - description of 4-5
- Physical address space
  - description of 8-2
  - physical address 8-2
- Physical addressing mode 8-1
- Physical memory
  - caching of memory accesses 8-3
  - description of 8-2
  - restrictions 8-2
- Pi 7-18
- Port segment descriptor 8-13
- Ports
  - description of 14-7
  - FIFO port 14-7
  - priority port 14-7, 14-8
  - segment descriptor 8-13
  - uses of 14-9
  - See also Communications port, Dispatch port
- PRCB
  - caching the PRCB in the processor 9-9
  - changing the PRCB 9-9
  - current process SS 9-8
  - description of 9-2, 9-5
  - dispatch port assigned to 14-10
  - dispatch port SS field 9-8, 14-10
  - fault resumption record 12-15
  - fault table pointer 9-9, 12-7
  - idle time field 9-9
  - initialization PRCB 9-14, 9-19
  - interrupt stack pointer 9-8
  - interrupt table pointer 9-8
  - modify processor controls IAC 11-15
  - multiprocessor preemption field 9-9

- pointers to system data structures 9-8
- procedure table pointer 4-11
- processor controls word 9-5
  - region 3 SS 9-8
  - resumption record field 9-9
  - store system base IAC 11-23
  - system procedure-table SS 9-9
  - system-error fault field 9-9, 12-16, 12-21
  - system-error fault record field 9-9, 12-16, 12-21
- See also Processor controls
- Preempt flag 13-5, 14-14, 15-4
- Preempt process IAC 11-16, 15-4, 15-5
- Preemption
  - description of 14-14
  - in a multiprocessor system 15-4
  - interim priority field in processor controls 15-5
  - multiprocessor preempt flag 9-5, 15-5
  - multiprocessor preemption action 15-6
  - nonpreempt limit field of processor controls 9-7, 15-5
  - preempt flag 15-4
  - preemption control in a multiprocessing system 15-4
  - write external priority flag 15-5
- Preemption IAC 14-14
- Prereturn trace
  - event flag 16-2
  - fault 12-2, 12-37
  - mode 16-5
  - mode flag 16-2
  - prereturn trace flag 4-5
- Preserved 1-3
- Previous frame pointer
  - See PFP
- Priorities 9-10
- Priority port 14-7
  - conditional receive message mechanism 14-17
  - description of 14-8
  - lock field 14-8
  - priority 14-8
  - queue head SS 14-9
  - queue state flag 14-8
  - queue status field 14-8
  - queue tail SS 14-9
  - receive message mechanism 14-17
  - send message mechanism 14-16
  - send service mechanism 14-18
- Procedure calls
  - branch and link 4-15
  - call/return mechanism 4-1
  - FP 4-3
  - local call 4-8
  - local registers 4-3
  - overview of 4-1
  - padding area 4-5
  - parameter passing 4-9
  - PFP 4-5
  - prereturn trace flag 4-5
  - procedure linking information 4-3
  - procedure stack 4-3
  - procedure table 4-11
  - return status field 4-5
  - RIP 4-6
  - saving of local registers 4-1
  - SP 4-3
  - supervisor call 4-14
  - supervisor stack 4-14
  - system call 4-10
  - user-supervisor protection model 4-13
- Procedure Stack
  - call/return mechanism 4-1
  - description of 4-3, 9-4
  - mapping of local registers to 4-7
  - process state 13-1
  - register save area 4-3, 4-7
  - stack frames 4-3
- Procedure table
  - procedure entry structure 4-11
  - segment descriptor 8-13
  - structure of 4-11
  - supervisor-stack-pointer entry 4-12

- Procedure table call 4-10  
    See also System call
- Procedure table segment descriptor 8-13
- Process
- address space 13-1
  - binding to processor 14-2
  - current process SS in PRCB 9-8
  - description of 9-1, 13-1
  - execution mode 4-13
  - flush process IAC 11-8
  - preempt process IAC 11-16
  - priority 13-4, 14-8
  - procedure stack 4-3
  - process controls 13-4
  - state 13-1, 13-4
  - timing 13-7
  - use of 13-2  
    See also PCB, Process management
- Process control block  
    See PCB
- Process control instructions 6-17
- Process controls
- changing of 13-8
  - description of 13-4
  - execution mode flag 13-4
  - Internal state field 12-11
  - next time slice field 14-5
  - preempt flag 13-5
  - priority field 13-4
  - process state 13-1
  - refault flag 13-5
  - residual time slice field 14-5
  - resume flag 13-5
  - state field 13-4
  - time-slice flag 13-5, 14-5
  - time-slice-reschedule flag 13-5, 14-5
  - timing flag 13-5, 14-5
  - trace enable flag 13-5
  - trace fault pending flag 13-5
- Process controls word  
    See Process controls
- Process management
- binding process to processor 13-11, 14-2
  - changing arithmetic controls 13-9
  - changing of process controls 13-8
  - changing the process notice field 13-9
  - concepts 14-2
  - dispatching 9-5, 14-2
  - execution time counting 14-6
  - explicit process dispatching 14-4
  - high-level process management facilities 14-1, 14-6, 15-3
  - instructions 6-17
  - kernel support for message passing 14-18
  - low-level process management facilities 14-1
  - messages 14-9
  - multiple-process management facilities, overview of 14-1
  - multiprocessor preemption 15-4, 15-6
  - overview of 13-1
  - PCB 13-2
  - physical addressing vs. virtual addressing 13-9
  - ports 14-7, 14-9
  - preemption 14-14
  - preemption control in a multiprocessing system 15-4
  - priority field 13-4
  - process controls 13-4
  - process faults 12-33
  - process handling in a single-process system 13-11
  - process states 14-2
  - process suspension 14-11
  - required software support for a single-process system 13-9
  - scheduling 9-5, 14-2
  - state transition actions 14-3
  - time-slice scheduling 14-5
  - timing 14-5  
    See also Automatic process dispatching, Messages and message passing, Process, Process synchronization



- Process notice
  - changing process notice field 13-9
  - check process notice IAC 12-13, 12-28
  - field in PCB 11-5, 13-6
- Process scheduling and dispatching
  - binding process to processor 14-2
  - description of 14-2
  - explicit process dispatching 14-4
  - See also Automatic process dispatching
- Process segment descriptor 8-13
- Process synchronization
  - description of 14-12
  - semaphores 14-12
- Process timing 14-5, 14-6
  - end-of-time-slice event 14-11
  - time slice fault 12-2, 12-33, 14-5, 14-6
  - time-slice scheduling 14-5
  - while handling a fault 12-13
  - while handling an interrupt 10-6
- Processor
  - freeze IAC 11-13
  - internal structure of C-1
  - modify processor controls IAC 11-15
  - multiprocessing system 9-1
  - multitasking system 9-1
  - overview of processor configurations 9-1
  - priorities 9-10
  - purge instruction cache IAC 11-17
  - reinitialize processor IAC 11-18
  - restart processor IAC 11-19
  - self test 9-21
  - single-task system 9-1
  - stop processor IAC 11-21
  - store processor IAC 11-22
  - store system base IAC 11-23
  - warmstart processor IAC 11-25
- Processor and process states
  - description of 9-10
  - idle state 9-11
  - idle-interrupted state 9-11
  - process-executing state 9-11
  - process-interrupted state 9-11
  - state field 9-7
  - stopped state 9-11
- Processor Control Block
  - See PRCB
- Processor controls
  - addressing mode flag 9-7
  - check dispatch port flag 9-7
  - description of 9-5
  - interim priority field 9-8, 15-5
  - modify processor controls IAC 9-10, 11-15
  - multiprocessor preempt flag 9-5, 15-5
  - nonpreempt limit field 9-7
  - nonpreempt limit field of processor controls 15-5
  - state field 9-7
  - write external priority flag 9-8, 15-5
- Processor controls word
  - See Processor controls
- Processor management
  - instructions 6-16
- Processor management facilities
  - faults 9-5
  - IACs 9-4
  - instruction list 9-2
  - interrupts 9-4
  - overview of 9-2
  - process scheduling and dispatching 9-5
  - system data structures 9-2
- Processor management, software requirements for 9-14
- Processor timing
  - duration of a timing 9-13
  - idle timing 9-13
- Programming environment
  - See Execution environment
- Protection faults 12-34
- Protection, support for 2-6
- Purge instruction cache IAC 11-17

## Q



- QNaN  
See NaNs
- Quad word, description of 5-5
- Queue linkage information in ports  
queue head SS 14-7  
queue state flag 14-7, 14-8  
queue status field 14-8  
queue tail SS 14-7
- Queue record, in PCB 13-6
- ## R
- Real number  
encodings 7-4  
system 7-1
- receive** 6-18, 10-5, 14-17, 17-109
- Refault flag 12-11, 12-12, 12-20, 12-21, 13-5
- reg, notation 17-2
- Regions 3-10  
gaps and boundaries 8-28  
making region boundaries transparent 8-28  
pointers for regions 0, 1, and 2 in PCB 13-6, 13-9  
region 3 SS in PRCB 9-8  
required at initialization 9-14  
spanning region boundaries 8-25  
typical address space structure 8-26
- Register indirect addressing modes  
description of 5-7
- Register indirect addressing modes, description of 5-7
- Register save area  
See Procedure stack
- Register scoreboarding 2-3, 3-5, C-7
- Registers  
addressing of 5-6  
floating-point registers 2-5, 3-2  
flush local registers IAC 11-7  
flush local registers instruction 6-16  
global registers 3-2  
local registers 3-2  
register model 3-2
- See also Floating-point registers,  
Global registers, Local registers
- Reinitialize processor IAC 11-18
- Remainder instructions 6-8
- remi, remo** 6-8, 17-111
- remr, remrl** 7-11, 7-17, 17-112
- Reserved 1-3
- RESET pin 9-21
- Residual time slice field 13-7, 14-5
- Restart processor IAC 3-6, 9-9, 9-23, 11-19, 12-22
- Resume flag 10-9, 12-11, 12-18, 12-19, 12-20, 12-21, 13-5, 13-8
- resumpres** 6-17, 10-5, 13-11, 14-2, 14-3, 14-4, 14-6, 17-115
- Resumption record field 9-9
- ret** 4-8, 6-15, 12-9, 12-18, 12-20, 16-5, 16-8, 17-116
- Return  
from local call 4-8  
from local system call 4-13  
from supervisor call 4-14
- Return instruction 6-15
- Return instruction pointer  
See RIP
- Return status field 12-18  
description of 4-5  
encoding of 4-5  
return from local system call 4-13  
return from supervisor call 4-14
- Return trace  
event flag 16-2  
fault 12-2, 12-37  
mode 16-5  
mode flag 16-2
- RIP 3-4, 3-6  
description of 4-6  
on a branch and link 4-15
- rotate** 6-9, 17-118
- Rotate instructions 6-9
- Rounding control  
See Floating-point rounding control  
field

**roundr, roundrl** 7-17, 17-119

## S

Saved IP, for fault 12-15

**saveprcs** 6-17, 13-11, 14-2, 14-4, 14-6,  
17-120

Scale factor in addressing, description of  
5-7

**scaler, scalerl** 7-19, 7-24, 17-121

**scanbit** 6-10, 17-123

**scanbyte** 6-11, 17-124

**schedprcs** 6-17, 14-10, 15-5, 17-125

Scoreboarding

See Register scoreboarding

Segment

bipaged region 8-12

description of 8-4

large segment table 8-14

paged region 8-12

port 8-13

procedure table 8-13

process 8-13

semaphore 8-15

simple region 8-11

small segment table 8-14

types 8-11

Segment descriptor

access status field 8-10

accessed flag 8-10

altered flag 8-10

base address field 8-9

bipaged region descriptor 8-12

cacheable flag 8-10

description of 8-9

invalid descriptor 8-16

large segment table descriptor 8-14

paged region descriptor 8-12

paging method field 8-11

port descriptor 8-13

procedure table descriptor 8-13

process descriptor 8-13

region descriptors 8-11

segment table descriptors 8-14

segment types 8-11

semaphore descriptor 8-15

simple region descriptor 8-11

size field 8-10

small segment table descriptor 8-14

valid flag 8-10

Segment length fault 8-21, 8-22, 8-24,  
12-2, 12-34, 17-74

Segment selector

See SS

Segment table

description of 8-8, 9-2

required at initialization 9-14

Self dispatching

See Automatic process dispatching

Self test, of processor 9-21

Semaphore

access action 14-13

count field 14-12

description of 14-12

high-level process management facilities  
14-1

instructions for handling semaphores

6-18, 14-13

lock field 14-12

semaphore queue tail SS 14-13

structure of 14-12

Semaphore segment descriptor 8-15

**send** 6-18, 14-16, 14-18, 15-5, 17-126

**sendserv** 6-18, 10-5, 14-10, 14-11, 14-18,  
14-19, 15-5, 17-128

Set breakpoint register IAC 11-20

Set, definition of 1-4

**setbit** 6-10, 17-130

Shift instructions 6-9

**shli, shlo, shrld, shri, shro** 6-9, 17-131

**signal** 6-18, 14-13, 15-5, 17-133

Significand, in floating-point format 7-2

Simple region segment descriptor 8-11

**sinr, sinrl** 7-18, 17-134

Size field 8-10

- SIZE lines 8-20
- Small segment table segment descriptor 8-14
- SNaN
  - See NaNs
- SP 3-4, 4-14
  - description of 4-3
- spanbit 6-10, 17-136
- sqrtr, sqrrl 7-17, 17-137
- SS
  - description of 8-5, 8-7
  - treatment of, depending on address translation mode 9-12
- st, stib, stis, stl, stob, stos, stq, stt 5-5, 6-5, 7-9, 17-139
- Stack
  - See Procedure stack
- Stack frame cache 4-3
  - flush local registers IAC 11-7
  - mapping to procedure stack 4-7
- Stack frame, definition of 4-3
- Stack pointer
  - See SP
- Standard faults 17-3
- STARTUP pin 9-21
- State field 9-7
- Sticky flags, definition of 3-7
- Stop processor IAC 11-21
- Store instructions 6-5
- Store processor IAC 9-10, 9-12, 11-22
- Store system base IAC 11-23
- String instructions 6-19
- Structural faults 12-36
- subc 6-8, 17-141
- subi, subo 6-8, 17-142
- subr, subrl 7-17, 17-143
- Subtract instructions 6-8
- Subtract with Carry Instruction 6-8
- Supervisor call 4-14
  - system call instruction 6-15
- Supervisor mode
  - See User-supervisor protection model
- Supervisor stack
  - description of 9-4
  - structure of 4-14
  - supervisor-stack pointer 4-12
- Supervisor trace
  - event flag 16-2
  - fault 12-2, 12-37
  - mode 16-5
  - mode flag 16-2
- Supervisor-stack pointer 4-12
- syncf 12-22, 17-145
- synld 6-20, 17-146
- synmov, synmovl, synmovq 6-20, 11-1, 11-3, 15-2, 15-3, 17-148
- System call
  - description of 4-10
  - mechanism of 4-10
- System data structures
  - description of 9-2
- System error fault
  - See System error interrupt
- System error interrupt
  - description of 10-8, 12-4
  - fault handling method 12-3
  - halt action 12-22
  - handling of 12-21
  - interrupt vector 248 12-4
  - relationship to halt 12-5
  - system-error fault field in PRCB 9-9, 12-16
  - system-error fault record field in PRCB 9-9, 12-16
  - system-error interrupt action 12-21
- System executive
  - Kernel 1-1
- System procedure table
  - description of 9-4
  - structure of 4-11
  - system call instruction 6-15
  - system procedure-table SS in PRCB 9-9

trace control flag 4-12  
 System procedure table SS 9-9  
 System-error fault field 9-9

## T

**tanr, tanrl** 7-18, 17-151  
 Task, description of 9-1  
 Terminology 1-3  
 Test instructions 6-14  
 Test pending interrupts IAC 11-24, 15-7  
**teste, testne, testl, testle, testg, testge, testno, testno** 6-14, 17-153  
 Tick 9-13  
 Time slice  
   See Process timing  
 time slice flag 10-9, 14-5  
 Time-slice-reschedule flag 14-5  
 Timing  
   See Process timing, Processor timing  
 Timing flag 10-5, 10-9, 14-5  
 TLB  
   description of 8-25  
   flush process IAC 11-8  
   flush TLB IAC 11-9  
   flush TLB page table entry IAC 11-10  
   flush TLB physical page IAC 11-11  
   flush TLB segment entry IAC 11-12  
 Trace control flag (in a procedure table)  
   12-19  
 Trace control flag (in system or procedure  
   table) 16-1, 16-3, 16-6  
 Trace control flag (in system procedure  
   table) 4-12  
 Trace controls  
   See Tracing  
 Trace controls field, in PCB 13-5  
 Trace enable flag 10-9, 12-12, 12-19, 13-5,  
   16-1, 16-3, 16-6, 16-7, 16-8  
 Trace fault handler procedure table 12-7,  
   12-8

Trace fault pending flag 10-9, 12-11, 13-5,  
   16-1, 16-3, 16-6, 16-7, 16-8  
 Trace flag (in return-status field of r0)  
   16-1, 16-3

## Tracing

branch trace mode 16-4  
 breakpoint registers 16-5  
 breakpoint trace mode 16-5  
 call trace mode 16-4  
 fault handlers, tracing with 16-8  
 handling multiple trace events 16-6  
 instruction trace mode 16-4  
 interrupt handlers, tracing with 16-7  
 modifying trace controls 16-2  
 overview of trace-control facilities  
   16-1  
 prereturn trace handling 16-7  
 prereturn trace mode 16-5  
 process state 13-1  
 return trace mode 16-5  
 signaling a trace event 16-6  
 software support required for tracing  
   16-1  
 supervisor trace mode 16-5  
 trace control flag (in system or procedure  
   table) 16-3  
 trace control on supervisor calls 16-3  
 trace controls 16-1  
 trace controls word 16-2  
 trace enable and mode flags 12-12  
 trace enable flag 16-3  
 trace event flags 16-2  
 trace fault handler 16-6  
 trace fault pending flag 16-3  
 trace faults 12-37, 16-1, 16-3, 16-5,  
   16-6  
 trace flag (in return-status field of r0)  
   16-3  
 trace handling action 16-7  
 trace mode flags 16-2  
 trace modes 16-4  
 tracing instructions 6-16

Translation look-aside buffer

See TLB

Triple word, description of 5-5

Type faults 12-39

Type mismatch fault 8-11, 12-2, 12-39,  
17-92, 17-115

## U

Unconditional branch instructions 6-13

Unordered

definition of 3-8

numbers 7-17

User-supervisor protection model

description of 4-13

mode switching 4-14

supervisor call 4-14

supervisor mode 4-13, 4-14

supervisor procedure 4-13, 4-14

user mode 4-13, 4-14

## V

Valid flag, description of 8-10

Virtual addressing mode

description of 8-1

load physical address instruction 8-25

SS translation action 8-21

virtual address translation action 8-22

Virtual memory faults 8-10, 12-40

Virtual memory management facilities

accessing system data structures 8-28

address translation action 8-21

inspect access instruction 6-20

load physical address instruction 6-20

operating system considerations 8-26

overview of 8-3

page table 8-5

page table and page table directory structures 8-18

page table directory 8-5

page tables and page table directories  
8-16

segment 8-4

*segment descriptor* 8-4, 8-9

segment selector 8-7

segment table 8-4, 8-8

segment table data structures 8-6

SS 8-5

TLB 8-25

typical address space structure 8-26

use of segments 8-5

See also Segment table

Virtual memory, support for 2-6

## W

**wait** 6-18, 10-5, 14-13, 17-155

Warmstart processor IAC 3-6, 9-9, 11-25

Words

addressing of 5-5

size 3-4

Write external priority flag 9-8, 15-5

## X

**xnor, xor** 6-10, 17-157

